The directed word graph of a text
Program Documentation

Hans Leiß

July 30, 2001

Abstract

The program described is an implementation of the algorithm by Blumer e.a. (TCS, 1985), *The smallest automaton recognizing the subwords of a text.*

We implement not the smallest automaton, but the index of all subwords of the text. The index of a text t is a deterministic automaton such that the state reached by reading a word w tells the set of end positions of the occurrences of w in the text t.

Contents

1 The automaton $\mathcal{A}_t$ accepting the subwords of a text t

1.1 The equivalence relation $\sim_t$

For each text $t \in \Sigma^+$, define the equivalence relation

$$u \sim_t v :\Longleftrightarrow \text{ endpositions of } u \text{ in } t = \text{endpositions of } v \text{ in } t$$

Let $[u]_t$ or just $[u]$ be the equivalence class of $[u]$ under $\sim_t$.

The form of the equivalence classes

For any two subwords v, w ending in the same position of t, either $v \leq_{suf} w$ or $w \leq_{suf} v$. Hence, if $u \in \mathit{Subw}(t)$, then class $[u]$ has the form

$$[u] = \{v \mid \min[u] \leq_{suf} v \leq_{suf} \max[u]\},$$

where $\min[u]$ is the shortest and $\max[u]$ the longest element of $[u]$. Note that $[u]$ can also be represented by a set of numbers,

$$[u] \simeq \{i \mid \text{ some occurrence of } u \text{ in } t \text{ ends at position } i \leq |t|\}.$$

These sets are partially ordered by $\subseteq$; without $\emptyset$, they form a *tree* under $\subseteq$, with $[\epsilon] = \{i \mid i \leq |t|\}$ as its root.

The automaton $\mathcal{A}_t$

Obviously, there are only finitely many such equivalence classes. It is easy to see that for all $a \in \Sigma$,

$$[u] = [v] \Rightarrow [ua] = [va],$$

so the equivalence classes for a finite automaton $\mathcal{A}_t = (Q, \Sigma, \delta, [\epsilon], F)$ where

$$\begin{array}{rcl} Q & = & \{[u] \mid u \in \Sigma^*\}, \\ i & = & [\epsilon], \\ \delta([u], a) & = & [ua], \\ F & = & \{[u] \mid u \text{ is a subword of } t\} \end{array}$$

This is an automaton such that $L(\mathcal{A}_t) = \{u \mid u \text{ is a subword of } t\}$. Note that there is a single state *reject* $\in Q$, containing all $u \in \Sigma^*$ that are not subwords of t.

2 The word graph $\mathcal{G}_t$ of the subwords of a text t

To construct the finite automaton $\mathcal{A}_w$ for a word w, in an on-line fashion, we need to enrich the automaton $\mathcal{A}_w$ by certain access-functions that make the construction of $\mathcal{A}_w = \mathcal{A}_{sa}$ from $\mathcal{A}_s$ fast. We expand the automata $\mathcal{A} = (Q, \Sigma, \delta, [\epsilon], F)$ to *wordgraphs* $\mathcal{G} = (\mathcal{A}, f, \mathit{max}, \mathit{sink})$, where:

1. a function *suffix* or $f : Q \to Q$, such that

$$f([u]) := \begin{cases} \mathit{reject}, & \text{if } u = \epsilon \text{ or } u \notin \mathit{Subw}(t), \\ [v], & \text{else, where } v \text{ is the longest suffix of } u \text{ with } v \not\sim u \end{cases}$$

2. $\mathit{max} \in \mathit{Nat}$, the maximal number used to number a state of $\mathcal{A}$; this need not be the number of $[s]_s$ in $\mathcal{A}_s$,

3. *sink* $\in Q$, the state $[s]_s$ for $\mathcal{A} = \mathcal{A}_s$.

To distinguish whether in $\delta([p], a) = [pa]$ the element pa is maximal in its class $[pa]$ or not, we distinguish two types of transitions: *primary* transitions $[p] - a \to [pa]$, for which we have $\max[pa] = (\max[p])a$, and *secondary* transitions, where this is not the case.

2.1 The signature `WORDGRAPH`

We fix an alphabet `token` $= \Sigma$ for the wordgraphs of texts $t \in \Sigma^*$ to be the SML-type `char` of characters.[1] A text given as a string can be translated to a `token`-sequence using SML's function `explode : string -> char list`.

The states of a wordgraph are identified by (non-negative) integers. Each word-graph has an initial state $[\epsilon]$, identified by `start:state`, and a single rejecting state $\Sigma^* \setminus \mathit{Subw}(s)$, identified by `reject:state`, whose numbers are common to the wordgraphs of all texts. The equivalence class $[u]_t$ identified by a number `q:state` is given by the set of *endpositions of* u in the text, obtained by `positions : state -> int list`.

In the type `dawg` of *directed accyclic word graphs*, we record the class $[t]_t$, identified by `sink:state`, the maximum state identifying number, `max:state`, the length of the text t, which is the end-position `pos:int` of the text t as prefix of a possibly on-going text, a list `tokens` of different input tokens seen. Moreover, we record the suffix-pointer $f(p) = \mathit{Suffix}(p) = \max\{w \in \mathit{Suffix}(p) \mid w \not\sim_t p\}$, more precisely, the function `suffix : state -> state` defined by

$$\mathit{Suffix}([p]) = [\max\{w \in \mathit{Suffix}(\max[p]) \mid w \not\sim_t p\}].$$

The function in the position-field yields to each state the list of end-positions that represents that state (in reverse order, so that adding the new position takes constant time when done with `cons`). Finally, we record the transition function $\delta_t : Q \times \Sigma \to Q$, in the field

```
trans : state * (token * bool) -> state.
```

The `bool` component in the argument type is used to distinguish whether in $\delta([p], a) = [pa]$ the element pa is maximal in its class $[pa]$ or not. I.e., we distinguish two types of transitions: *primary* transitions $q - a \to q'$, for which we have $\max q' = (\max q)a$, and *secondary* transitions, where this is not the case. We use labels $(a, 1)$ to mark primary and $(a, 0)$ to mark secondary transitions.

$\langle$*dawg.sml*$\rangle\equiv$

```
signature WORDGRAPH =
    sig
        type token = char
        type state = int

        val start  : state
        val reject : state

        type dawg = {sink : state,
                     states : state list,
                     max : int,
                     pos : int,
                     tokens : token list,
                     positions : state -> int list,
                     suffix : state -> state,
```

[1] Later, we may relax `token` to be an arbitrary equality type, but displaying of graphs depends on a translation from tokens to strings.

```
                    trans : state * (token * bool) -> state}

    val mkDawg : string -> dawg
    val updateDawg : token * dawg -> dawg
end
```

To create the wordgraph for the string t, use `mkDawg t`; to update a given wordgraph `G` of a text s to the one for the text sa, use `updateDawg (#"a",G)`, where `#"a"` is SML's notation for the character printed as a.

We do not add fields for `start` and `reject` in the type `dawg`, as these states are the same for all wordgraphs. (To display these states in a distinguished way, we may use their values `Wordgraph.start` and `Wortgraph.reject`.)

2.2 The structure `Wordgraph:WORDGRAPH`

This structure contains the explicitly given wordgraph of the empty text on the one hand and update-function to create the wordgraph of a text sa from the wordgraph of s and the single letter a.

The meaning ot the types is as explained for the signature `WORDGRAPH`.

$\langle$*dawg.sml*$\rangle$+≡

```
structure Wordgraph =
   struct
      type token = char
      type state = int

      val start = 0
      val (SOME (reject : state)) = Int.maxInt

      type dawg = {sink : state,
                   states : state list,
                   max : int,
                   pos : int,
                   tokens : token list,
                   positions : state -> int list,
                   suffix : state -> state,
                   trans : state * (token * bool) -> state}
```

2.2.1 The word graph $\mathcal{G}_\epsilon$ of the empty text

First, the word graph of the empty text, which clearly is as follows:

$\langle$*dawg.sml*$\rangle$+≡

```
      val initG : dawg =
         {sink = 0,
          states = [0],
          max = 0,
          pos = 0,
          positions = (fn q => if q=0 then [0] else []),
          tokens = [],
          suffix = (fn q => reject),
          trans = (fn (q, (a:token,b:bool)) => reject)}
```

2.2.2 Extending the wordgraph $\mathcal{G}_s$ to $\mathcal{G}_{sa}$

We represent an equivalence class or state $[u]_s$ by a number q and the list of end positions of u in s, given as `positions(q) : int list`. When constructing $\mathcal{G}_{sa}$ from $\mathcal{G}_s$, $[u]_s$ changes to (a subset) $[u]_{sa}$, i.e. the set of end positions may increase (by $|sa|$).

In the following, for partial functions g on states we simulate $g(q) = \mathit{undefined}$ by $g(q) = \mathit{reject}$. We use two auxiliary functions independent of $\mathcal{G}_s$:

⟨*dawg.sml*⟩+≡

```
fun updateF ((arg,value),f) =
   (fn x => if x=arg then value else (f x))

fun add a [] = [a]
  | add a (b::bs) = if a = b then b::bs else b::(add a bs)
```

In the following, `G:dawg` is the wordgraph of a text `s`. We want to extend it to the wordgraph of the text `sa`, i.e. the texts `s` extended by a single token `a`.

⟨*dawg.sml*⟩+≡

```
fun updateDawg (a,G : dawg) : dawg =
   let
```

On reading the new token `a`, we have to increase the position to $|sa|$ and need a number `newSink` for the new state $[sa]_{sa}$:

⟨*dawg.sml*⟩+≡

```
      val newSink = (#max G) + 1
      val newPos = (#pos G) + 1
      val newTokens = add a (#tokens G)
```

The only new states besides $[sa]_{sa}$ or `newSink` that possibly have to be introduced are states $[pa]_{sa}$ where $p \in \mathit{Suffix}(s)$. If $p \sim_s p'$, then $pa \sim_{sa} p'a$, so we only have to care about suffixes p of s that lie on the *suffix-chain* from the sink-state $[s]_s$, i.e. `(#sink G)`.

⟨*dawg.sml*⟩+≡

```
      fun suffixChain q =
         if not (hasTransition (q,a)) andalso not (q = reject)
            then q::(suffixChain (#suffix G q))
         else []

      and hasTransition (q,a) =
         hasPrimaryTransition (q,a) orelse hasSecondaryTransition (q,a)
      and hasPrimaryTransition (q,a) =
         not (#trans G (q,(a,true)) = reject)
      and hasSecondaryTransition (q,a) =
         not (#trans G (q,(a,false)) = reject)

      val chain = suffixChain (#sink G)
```

For each suffix p of s that lies on the suffix-chain, we have to add a new transition $[p]_{sa} \xrightarrow{a} [sa]_{sa}$, because there is no occurrence of pa in s. Clearly, pa is maximal in $[pa]_{sa}$ iff $p = s$, so in this case the transition has to be a primary one.

⟨*dawg.sml*⟩+≡

```
      fun toNewSink p = if p = (#sink G) then ((p,(a,true)),newSink)
                        else ((p,(a,false)),newSink)
```

```
            val newTrans =
               foldl updateF (#trans G) (map toNewSink chain)
```

(Note that although the transition starts in $[p]_{sa}$, we can use the state identifier of $[p]_s$ instead, and later care about possibly changing its position set.) More changes to the transitions will have to be made in some cases later on.

The suffix-chain contains at least the state (`#sink G`), i.e. the class $[s]_s$, since $\mathcal{G}_s$ has no non-rejecting a-transition from $[s]_s$. Consider the final element on the suffix-chain.

$\langle$*dawg.sml*$\rangle$+≡

```
         in
            if (List.last chain) = start then
```

If it is `start`, then a does not occur in s, so the modifications of $\mathcal{G}_s$ that have to be made consist in adding[2] `newPos` to the positions of `newSink` and `start`, and adding transitions from every suffix of s to $[sa]_{sa}$:

$\langle$*dawg.sml*$\rangle$+≡

```
               {sink = newSink,
                states = add newSink (#states G),
                max = newSink,
                pos = newPos,
                tokens = newTokens,
                positions = (fn q => if q = start
                                        then newPos :: (#positions G q)
                                     else if q = newSink then [newPos]
                                          else #positions G q),
                suffix = (fn q => if q = newSink then start else (#suffix G q)),
                trans = newTrans}
```

Otherwise, the last element on the suffix-chain is not $[\epsilon]$ = `start`, and the text s contains an occurrence of a. Then the `#suffix` of the last element of the suffix-chain, say p, has an a-transition in $\mathcal{G}_s$.

$\langle$*dawg.sml*$\rangle$+≡

```
            else
               let
                  val p = #suffix G (List.last chain)
               in
```

We distinguish whether pa is maximal in $[pa]_s$ or not. If it is, $[p]_s$ has a primary a-transition. Then pa is the longest suffix of sa such that $pa \not\sim_{sa} sa$, so we have to set the suffix-pointer of $[sa]_{sa}$ to point to $[pa]_s$:

$\langle$*dawg.sml*$\rangle$+≡

```
                  if hasPrimaryTransition (p,a) then
                          (*                       delta([r],a) = [sa] *)
                     let
                        val pa = #trans G (p,(a,true))
                        fun newSuff q = if q = newSink then pa else (#suffix G q)
```

The new position has to be propagated along the suffix-links to the root: for all the suffixes p of s, pa has an occurrence at the `newPos`, i.e. at $|sa|$. Here we have to follow the suffix-link up to the root, not just to a node with an out-going a-transition.[3]

[2] Here and later, it would be faster to increase the state-list by `newSink::(#states G)`, but we use `(add newSink (#states G))` to generate an ordering of states that is nicer in the graphical representation.

[3] This step and a similar one below seems to corrupt the $O(|t|)$ time bound of the algorithm.

⟨*dawg.sml*⟩+≡

```
                fun newPositions q =
                   let
                      fun rootPath q acc =
                         if q = start then (q::acc)
                         else rootPath (newSuff q) (q::acc)

                      fun pos' s = if s = newSink then []
                                   else (#positions G s)
                      fun upd (q,f) = updateF ((q, newPos::(pos' q)),f)
                      val g = (foldr upd pos' (rootPath newSink []))
                   in
                      g q
                   end
             in
                {sink = newSink,
                 states = add newSink (#states G),
                 max = newSink,
                 pos = newPos,
                 tokens = newTokens,
                 positions = newPositions,
                 suffix = newSuff,
                 trans = newTrans}
             end
```

Otherwise, if p has not a primary a-transition, it must have a secondary a-transition, because otherwise it would be on the suffix-chain of $\mathcal{G}_s$. Hence we know that $\max[pa]_s$ is longer than pa. We have to split the equivalence class

$$[pa]_s = \{\min[pa]_s, \ldots, pa, vpa, \ldots, \max[pa]_s\}$$

into two parts,

$$[pa]_{sa} = \{\min[pa]_s, \ldots, pa\} \qquad \text{and} \qquad \{vpa, \ldots, \max[pa]_s\}.$$

We introduce a new state number for the first part and use the number of $[pa]_s$ for the second part. Considered as a position set, the first part is just $[pa]_s \cup \{|sa|\}$. Then we have to adjust the transition- and suffix-functions.

⟨*dawg.sml*⟩+≡

```
          else
             let
                val pa = #trans G (p,(a,false)) (* [max[pa]_s]_sa *)
                val newState = newSink + 1      (* [pa]_sa *)

                fun suffixChain' q =
                   if (hasTransition (q,a))
                      andalso not (q = reject)
                      then q::(suffixChain' (#suffix G q))
                   else []
                val chain' = suffixChain' (#suffix G p) (* ! *)

                val newTrans =
                   let
```

First, all edges leaving the $[pa]_s$ in $\mathcal{G}_s$ need to be copied to edges leaving the new state $[pa]_{sa}$ and ending in the same state as before:

$\langle$*dawg.sml*$\rangle$+≡

```
fun outgoing (b,f) =
   if hasPrimaryTransition (pa,b)
      then updateF (((newState,(b,false)),
                    #trans G (pa,(b,true))),f)
   else if hasSecondaryTransition (pa,b)
           then updateF (((newState,(b,false)),
                             #trans G (pa,(b,false))),f)
         else f
val outTr1 = foldr outgoing (#trans G) (#tokens G)
```

But when we have $[pa]_s =$`(#sink G)`, the new (primary) transition $[pa]_s \overset{a}{\Longrightarrow} [sa]_{sa}$ is not covered by `hasTransition`, so we need to copy it separately:

$\langle$*dawg.sml*$\rangle$+≡

```
val outTr2 = updateF (((#sink G,(a,true)),newSink),outTr1)

fun outgoingNew (b,f) =
   if pa = #sink G andalso b = a
      then updateF (((newState,(b,false)), newSink), f)
   else f

val outTr = foldr outgoingNew outTr2 (#tokens G)
```

Next, all edges leading to $[pa]_s$ from states $[r]_s$ where r was a suffix of p must be redirected to lead to the new state $[pa]_{sa}$;

$\langle$*dawg.sml*$\rangle$+≡

```
fun incoming (q,f) =
   if f (q,(a,false)) = pa then
      updateF (((q,(a,false)),newState),f)
   else f
val inTr1 = foldl incoming outTr chain'
```

Replace the secondary edge $[p]_s \xrightarrow{a} [pa]_s$ by a primary edge $[p]_s \overset{a}{\Longrightarrow} [pa]_{sa}$:

$\langle$*dawg.sml*$\rangle$+≡

```
val inTr2 = updateF (((p,(a,false)),reject),inTr1)
val inTr = updateF (((p,(a,true)),newState),inTr2)
```

Finally, for suffixes p of s we need transitions $[p]_s \xrightarrow{a} [sa]_{sa}$ to the new sink state as in the other cases:

$\langle$*dawg.sml*$\rangle$+≡

```
in
   (foldl updateF inTr (map toNewSink chain))
end
```

The new suffix-function is as follows: (have to check if this is o.k.!)

$\langle$*dawg.sml*$\rangle$+≡

```
fun newSuff q =
   if q = newState then #suffix G pa
   else if q = pa then newState
```

```
                                  else if q = newSink then newState
                                       else #suffix G q
```

The equivalence classes $[ra]_{sa}$ and $[ra]_s$ are the same for r in the suffix chain of p. But for the representation by position sets we have to propagate the new end position along the suffix links.

$\langle$*dawg.sml*$\rangle$+≡

```
                          fun newPositions q =
                             let
                                fun rootPath q acc =
                                   if q = start then (q::acc)
                                   else rootPath (newSuff q) (q::acc)

                                fun pos' s = if s = newSink then [newPos]
                                             else if s = newState then
                                                 newPos::(#positions G pa)
                                                  else (#positions G s)

                                fun upd (q,f) = if q <= #max G then
                                   updateF ((q, newPos::(pos' q)),f) else f
                                val g = (foldr upd pos' (rootPath newSink []))
                             in
                                g q
                             end
```

In the list of states, we add the `newState` near to the state it was obtained from, so that the displaying of the graphs is a bit nicer. (Of course, using `(newState::newSink::(#states G))` would be more efficient.)

$\langle$*dawg.sml*$\rangle$+≡

```
                          fun replace (n,ms) (q::qs) acc =
                             if q = n then (rev qs) @ (ms @ (rev acc))
                             else replace (n,ms) qs (q::acc)
                          val newStates =
                             replace (pa,[newState,pa]) (newSink :: (rev (#states G))) []
                       in
                          {sink = newSink,
                           states = newStates,
                           max = #max G + 2,
                           pos = newPos,
                           tokens = newTokens,
                           positions = newPositions,
                           suffix = newSuff,
                           trans = newTrans}
                       end
                 end
           end
        fun mkDawg text = foldl updateDawg initG (explode text)

        fun state (G:dawg) w =
            let
                fun hasPrimaryTransition (q,a) =
                    not (#trans G (q,(a,true)) = reject)
                and hasSecondaryTransition (q,a) =
                    not (#trans G (q,(a,false)) = reject)
```

```
            fun delta (a,q) =
                if hasPrimaryTransition (q,a) then (#trans G (q,(a,true)))
                else (#trans G (q,(a,false)))
        in
            foldl delta start (explode w)
        end
    and occurs G w = #positions G (state G w)
end
```

3 Graphical presentation of a word graph

To display the subword-index of a text, or the corresponding graph incuding the suffix-pointers, we produce a file `<file>.dot` in the format of `AT&T`'s graph-viewing package `graphviz/dot`. This `.dot`-file is then translated to a *Postscript* file `<file>.ps`, which can be viewed at the srceen.

To help to understand the algorithm, we also produce a `.dot`-file containing the sequence of graphs needed to construct the wordgraph of a given text.

3.1 The signature ppWORDGRAPH

⟨*display.sml*⟩≡

```
signature ppWORDGRAPH =
    sig
        structure W : WORDGRAPH

        type dawg = W.dawg
        type state = W.state
        type token = W.token

        type dotoption = string
        val default : dotoption

        val displayIndex : string -> dotoption -> OS.Process.status
        val displayDawgs : string -> dotoption -> OS.Process.status
        val displayGraphs : dawg list -> dotoption -> OS.Process.status

        val showIndex : string -> dotoption -> string -> OS.Process.status
        val showDawg : string -> dotoption -> string -> OS.Process.status
        val showGraphs : dawg list -> string -> string -> OS.Process.status

        val viewgraph : string -> OS.Process.status
    end
```

In these functions, the first argument of type `string` is the input text. The `display`-versions write to files `"wordgraph.tmp.dot"` and `"wordgraph.tmp.ps"`, and similar for `wordindex`.

Options for the underlying `dot`-program can be provided by an additional argument; for the format, see the man-page of `dot`. u If the picture is too large, use size-bounds and ratio-values to bring it to a reasonable size. The `dot`-program needs parameters `size=x,y` with values for x- and y-dimension in *inches*, and `ratio=x.y` for the ratio between x- and y-dimension. We use `"center=true;\nsize=\"6.0,10\";\nratio=2.3"` as default.

The `show`-versions allow you to provide explicit filenames (zpinstead of `wordindex.tmp` and `wordgraph.tmp`) and then use the `<filename>.ps`-file for inclusion in LaTeX-documents. The filename (without extensions `.dot` or `.ps`) is given in the last argument of type `string`.

The function `viewgraph` translates `<file>.dot` to `<file>.ps` and calls the previewer `ghostview`.

3.2 The structure ppWordgraph

$\langle$*display.sml*$\rangle$+≡

```
structure ppWordgraph : ppWORDGRAPH =
    struct
        structure W = Wordgraph

        type dawg = W.dawg
        type token = W.token
        type state = W.state

        type pda = {start : state list,
                    final : state list,
                    trans : (state * (token * bool) * state) list}

        fun dawgToPartialAutomaton (G:dawg) : pda =
            let
                val states = #states G
                val tokens = #tokens G
                val r = W.reject
                val accepting = List.filter (fn q => not (q = r)) states

                val Trs = ref [] : (int * (token * bool) * int) list ref

                fun addTr (a,p:state) =
                    let
                        val q = #trans G (p,a)
                    in
                        if p = r orelse q = r then ()
                        else Trs := ((p,a,q)::(!Trs))
                    end

                val _ = List.app (fn p =>
                            (List.app (fn a => addTr ((a,true),p)) (#tokens G)))
                        accepting
                val _ = List.app (fn p =>
                            (List.app (fn a => addTr ((a,false),p)) (#tokens G)))
                        accepting
                val transitions = (!Trs)
            in
                {start = [0],
                 final = map (fn s:state => s) accepting,
                 trans = transitions}
            end

        fun start ({start = S,...}:pda) = S
        and final ({final = F,...}:pda) = F
        and trans ({trans = T,...}:pda) = T

        fun node i = "node"^Int.toString i
```

```
fun toString (p::q::rest) = Int.toString p^","^(toString (q::rest))
  | toString (p::nil) = Int.toString p
  | toString nil = " "

fun drawIndex (M,Positions) os =
    let
        fun write w = TextIO.output (os, w)

        fun defineNode (i,oldstates) =
            write (node i^" [label = \""^(toString oldstates)^"\"];\n")

        fun drawTransition (i,(a,_),j) =
            write (node i ^ " -> " ^ (node j) ^
                   " [label = " ^ (Char.toString a) ^ "];\n")

        fun drawInitialState i = write (node i ^ " [shape = diamond];\n")

    in
        app defineNode Positions;
        app drawTransition (trans M);
        app drawInitialState (start M)
    end

fun drawIndex' (M,Positions) os =
    let
        fun write w = TextIO.output (os, w)

        fun drawInitialState i = write (node i ^ " [shape = diamond];\n")

        fun defineNode (i,oldstates) =
            write (node i^" [label = \""^(toString oldstates)^"\"];\n")

        fun drawTransition (i,(a,false),j) =
            write (node i ^ " -> " ^ (node j) ^
                   " [style = dashed, label = " ^ (Char.toString a) ^ "];\n")
          | drawTransition (i,(a,true),j) =
            write (node i ^ " -> " ^ (node j) ^
                   " [style = solid, label = " ^ (Char.toString a) ^ "];\n")
    in
        app defineNode Positions;
        app drawTransition (trans M);
        app drawInitialState (start M)
    end

fun drawSuffixLinks (G : dawg) os =
    let
        val r = W.reject

        fun write w = TextIO.output (os, w)

        fun drawLinks (p::ps) =
            let
                val q = #suffix G p
            in
```

```
                if p = r orelse q = r then drawLinks ps
                else (write (node p ^ " -> " ^ (node q) ^
                           " [color = blue, style=dotted];\n");
                      drawLinks ps)
                end
          | drawLinks [] = ()
    in
        drawLinks (#states G)
    end

fun drawSinkAndNewNode (G : dawg) os =
    let
        fun write w = TextIO.output (os, w)
    in
        write (node (#sink G)^" [color = orange];\n");
        if (#max G > #sink G) then write (node (#max G) ^" [color = red];\n") else ()
    end

fun statesWithPositions (G : dawg) =
    map (fn q => (q, rev (#positions G q)))
    (List.filter (fn q => not (q = W.reject)) (#states G))

type dotoption = string

val default =
   "center=true;\nsize=\"6.0,10\";\nratio=2.3"  (* \n size=\"7.5,14\"; *)
```

To display or print the graphs, translate the `.dot`-description to a `.ps`-file (in ordinary Postscript, since `dot` has no encapsulated Postscript) and then call a previewer (resp. print the `.ps`-file):

⟨*display.sml*⟩+≡

```
fun viewgraph filename =
   (OS.Process.system
    ("dot -Tps " ^ filename ^ ".dot -o " ^ filename ^ ".ps ");
    OS.Process.system ("ghostview -magstep -2 " ^ filename ^ ".ps &"))

fun displayIndex text options =
   showIndex text options "wordindex.tmp"

and showIndex text options filename =
   (dotIndex (W.mkDawg text) options filename; viewgraph filename)

and dotIndex G options filename =
    let
        val os = TextIO.openOut (filename ^ ".dot")
        fun write w = TextIO.output (os, w)
        val (A,Pos) = (dawgToPartialAutomaton G, statesWithPositions G)
    in
       write ("digraph SubwordIndex {" ^
              (if options = "" then default else options) ^ ";\n");
       drawIndex (A,Pos) os;
       write "}\n";
       TextIO.closeOut os
    end
```

By default, we write to files "wordgraph.tmp.dot" and "wordgraph.tmp.ps", with options provided by an additional argument of type `string`; for the format, see the man-page of `dot`.

If the picture is too large, use size-bounds and ratio-values to fit it to a reasonable size. The `dot`-program needs parameters `size=x,y` with values for x- and y-dimension in *inches*, and `ratio=x.y` for the ratio between x- and y-dimension.

Instead of default-filenames, one can give explicit filenames and then use the `<filename>.ps`-file to include in LaTeX-documents. To provide an explicit filename, use the versions `showIndex` and `showDawg`.

⟨*display.sml*⟩+≡
```
fun displayGraphs Gs options =
    showGraphs Gs options "wordgraph.tmp"

and showGraphs Gs options filename =
    (dotGraphs Gs options filename; viewgraph filename)

and dotGraphs Gs options filename =
    let
        val os = TextIO.openOut (filename ^ ".dot")
    in
        app (fn G => dotGraph G os options) Gs;
        TextIO.closeOut os
    end
and dotGraph G os options =
    let
        fun write w = TextIO.output (os, w)
        val (A,Pos) = (dawgToPartialAutomaton G, statesWithPositions G)
    in
        write ("digraph Wordgraph"^(Int.toString (#pos G)) ^ "{" ^
              (if options = "" then default else options) ^ ";\n");
        drawIndex' (A,Pos) os;
        drawSuffixLinks G os;
        drawSinkAndNewNode G os;
        write "}\n\n"
    end

and showDawg text options filename =
    showGraphs [W.mkDawg text] options filename
```

To display the history of the construction of a wordgraph, use the following function `displayDawgs`. It writes all the graphs to a single dot-file, so that the previewer allows you to go back and forth between the graphs using up- and down-keys.

⟨*display.sml*⟩+≡
```
local
   fun dawgs (c::cs) (G::Gs) =
       dawgs cs (W.updateDawg(c,G) :: (G::Gs))
     | dawgs (c::cs) [] =
       dawgs cs [W.updateDawg(c,W.initG),W.initG]
     | dawgs [] [] = [W.initG]
     | dawgs [] acc = rev acc
in
   fun displayDawgs text options =
```

```
            displayGraphs (dawgs (explode text) []) options
        end
    end
```

4 Makefile

Initially, I used the structure `ppFm` for displaying finite automata to produce the pictures for word-graphs, but all this has been removed because of the `bool`-part of the transition labels, and in order to make this program independent of the automata package.

The `sml`-code is extracted from the present file using `noweb`, Norman Ramsey's literate programming tool. SML of New Jersey's compilation manager uses `noweb` to extract the code from the following file `sources.cm`

⟨*sources.wordgraph.cm*⟩≡
```
Group is

    dawg.sml@wordgraph.nw
    display.sml@wordgraph.nw
    test.sml@wordgraph.nw
```

To improve the documentation, first load an auxiliary file by calling `- use "../NW.sml;"` in SML. The file `../NW.sml` declares functions like `NW.latex()` which translates the current file into legal LaTeX and runs the translated file to produce a `.dvi`-version.

The compilation manager of SML of New Jersey weaves fresh versions of the `sml`-program from this file when you call `CM.make();`.

5 Examples

⟨*test.sml*⟩≡
```
structure W = Wordgraph;
structure PPW = ppWordgraph;
```

To display the subword-index, the subword-graph, or the history of construction of the subword-graph of a string with some 15 states, we use the following functions, with default values for the `dot`-program:

⟨*test.sml*⟩+≡
```
fun ppIndex string  = PPW.displayIndex string ""
and ppGraphs string = PPW.displayDawgs string "";
```

For example, the graphs in the paper by Blumer e.a. (here with suffix-links added) are obtained by

⟨*test.ignored*⟩≡
```
    PPW.displayDawgs "abcbcd" "";
```

If the number of states is too large, we may determine the size (in inches) of the picture and the ratio between horizontal and vertical dimension by adding parameters for the underlying `dot`-program:

⟨*test.ignored*⟩+≡
```
PPW.displayIndex "abcac" "size=\"5.,6.\" center=true";
PPW.displayDawgs "abcac" "size=\"5.,6.\" center=true";
```

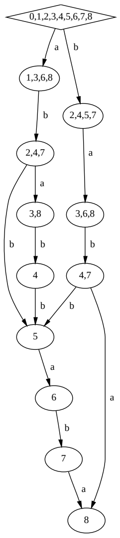
```
(* Demonstrating Fall II.2.b) slides: (pa is not maximal in [pa]_s) *)
   PPW.displayDawgs "baaaaaa" "";
```

To produce a `.ps`-file of the index or graph, use the following:

⟨*test.ignored*⟩+≡

```
  PPW.showDawg "ababbaba" "size=\"7.,6.\" center=true" "wordgraph.ababbaba";
  PPW.showIndex "ababbaba" "size=\"7.,6.\" center=true" "wordindex.ababbaba";
```

The latter generates the following graph:



It has been included using `depsfig{file=wordindex.ababbaba.ps,height=7cm}`.

6 Efficiency

Although the algorithm of Blumer e.a. constructs the wordgraph of t in $O(|t|)$ steps, the program gets very slow when t is 'longer', already for example for $(ab)^4(ba)^4$. The reason for this seems to be:

1. The program builds the *wordindex*, while the $O(|t|)$ bound seems to be for the *wordgraph* only. To obtain the index, for each added node `newSink` we traverse all the suffix-links from `newSink` to `start` and add position $|t|$ to the position sets of the states we meet. Blumer e.a. show that the traversal of the suffix-*chains* is $O(|t|)$, and they remark that traversing all the suffix-*links* needs $O(|t|^2)$ steps. Therefore, it seems that constructing the wordindex needs $O(|t|^2)$ instead of $O(|t|)$ steps.

2. Besides this, we might use better data structures and work with updating a given graph rather than building new ones. In particular,

   (a) we might omit the field `states : state list` in the type `dawg`, which is only used to map a function over the list of states and to have a better layout of the graphical representation by the `dot`-program. Clearly, we can do with `max`, the maximal state number, for the first purpose, and can therefore remove the state list at the cost of a slightly less natural graphical presentation.

(b) we might replace functions and their updating in a different way. The program does not assume to know the text length and represents functions by SML-functions. This may be good for small functions, but it is not clear what it means when the function is updated several times (for the same argument).

3. Use the profiler to see where most of the time is spent. The code can be instrumented and then applied to an example.

⟨*test.sml*⟩+≡

```
fun example () =
    let
        val G = W.mkDawg "abababbababa";
    in
        #positions G 4
    end
```

Using `timing.sml`, we see that most of the time is needed for displaying the graphs, not for constructing them:

Example	CPU time	GC time	Real time
`PPW.displayDawgs "abababbababa" "";`	40.260	0.020	66.990
`val G = W.mkDawg "abababbababa";`	0.060	0.000	19.824
`W.mkDawg` $(ab)^{10}(ba)^{10}$	0.230	0.000	11.030
`dags (explode` $(ab)^{10}(ba)^{10}$`)`	0.250	0.000	

It takes quite a while to do the last example with `displayDawgs`. When adding the function `occurs:string -> dawg -> int list` that tells where a string ends in a text, I realized that the call to `#positions G` is very slow, hence also the displaying of the index or graph. The reason seems to be the updating of functions used to define `#positions G`.

So it seems that we better use `type state = {nr : int, positions : int list ref}` and update the position sets without changing the state identity. But this modifies previous graphs in a develpoment of $\mathcal{G}_t$ as well, which makes it a little bit more complicated to print the history $\mathcal{G}_\epsilon, \ldots, \mathcal{G}_t$ of the construction of $\mathcal{G}_t$. However, the costs of the current version of `#positions` may make this chance worthwhile.

I didn't use `type state = {nr : int, positions : int list}`, because this would make updating the transition and suffix-functions more complicated, as we had to modify the arguments to these functions as well.

The profiler shows that indeed `upd` and `updateF` are used incredibly often!

```
- profileExpl();
GC #1.3.19.38.209.20167:   (0 ms)
 %time cumsec  #call  name
 17.39    .28 437473  Wordgraph.<tempStr>.updateDawg.updateDawg.newPositions.newPositions.upd
 13.66    .50 437473  Wordgraph.<tempStr>.updateDawg.updateDawg.newPositions.newPositions.roo
 13.66    .72 320423  Wordgraph.<tempStr>.updateF.updateF.anon
 10.55    .89 441582  Wordgraph.<tempStr>.updateF.updateF
  9.93   1.05 141467  Wordgraph.<tempStr>.updateDawg.updateDawg.newPositions.newPositions
  9.31   1.20 437473  Wordgraph.<tempStr>.updateDawg.updateDawg.newPositions.newPositions.roo
  4.96   1.27 405190  Wordgraph.<tempStr>.updateDawg.updateDawg.anon
  3.72   1.33 402547  Wordgraph.<tempStr>.updateDawg.updateDawg.newSuff.newSuff
  3.72   1.39 500532  Wordgraph.<tempStr>.updateDawg.updateDawg.newPositions.newPositions.pos
  3.10   1.43 184528  Wordgraph.<tempStr>.updateDawg.updateDawg.anon.anon
```

6.1 Measuring time and profiling

Profiling

The profiler needs instrumented code to measure time and space resources. So we first have to recompile the sources with profile-mode set to true:

⟨*instrumentCode*⟩≡
```
fun instrumentCode makefile =
    (Compiler.Profile.setProfMode true;
     Compiler.Profile.setTimingMode true;
     OS.Process.system
     "touch dawg.sml display.sml set.sml";
     CM.make' makefile)

and deinstrumentCode makefile =
    (Compiler.Profile.setProfMode false;
     Compiler.Profile.setTimingMode false;
     OS.Process.system
     "touch dawg.sml display.sml set.sml";
     CM.make' makefile)

fun instrumentExpl () = instrumentCode "sources.cm"
and deinstrumentExpl () = deinstrumentCode "sources.cm"
```

Having recompiled the system with instrumented code, we can apply the profiler. The example in `<test>` is taken from `wordgraph`.

⟨*profiling the example*⟩≡
```
fun profileExpl () =
  (Compiler.Profile.reset();
   example(); (* see wordgraph *)
   (* Compiler.Profile.report TextIO.stdOut; *)
   Compiler.Profile.reportAll TextIO.stdOut;
   Compiler.Profile.reportData ();
   Compiler.Profile.reset())
```

To switch off the profiler, use

⟨*no profiling*⟩≡
```
fun noProfile () =
    (Compiler.Profile.setProfMode false;
     Compiler.Profile.setTimingMode false)
```

To run the example with instrumented code, apply the following:

⟨*profiling and timing the example*⟩≡
    ⟨*profiling the example*⟩
    ⟨*timing the example*⟩

To run a test on the example and then return to non-instrumented code, do:

⟨***⟩≡
  ⟨*instrumentCode*⟩
  ⟨*profiling and timing the example*⟩

⟨*no profiling*⟩

Measuring time

Measuring time needs structure Timer from timer.sml and depends on the SML/NJ compiler. We do not want to measure compile time, so we use a function which we run, if we measure time.

⟨*timing the example*⟩≡
```
fun timeExpl () =
   let
      val a = CPUTiming.start();
      val _ = example()
   in
      (CPUTiming.format(CPUTiming.check a))
   end
```

Now do `timeExpl(); profileExpl(); noProfile();`