Computing all ℓ-cover automata fast[⋆]

Artur Jeż[1,⋆⋆] and Andreas Maletti[2,⋆⋆⋆]

[1] Institute of Computer Science, University of Wrocław
ul. Joliot-Curie 15, 50–383 Wrocław, Poland
aje@cs.uni.wroc.pl

[2] Institute for Natural Language Processing, Universität Stuttgart
Azenbergstraße 12, 70174 Stuttgart, Germany
andreas.maletti@ims.uni-stuttgart.de

Abstract. Given a language L and a number ℓ, an ℓ-cover automaton for L is a DFA M such that its language coincides with L on all words of length at most ℓ. It is known that an equivalent minimal ℓ-cover automaton can be constructed in time $\mathcal{O}(n \log n)$, where n is the number of states of M. This is achieved by a clever and sophisticated variant of Hopcroft's algorithm, which computes the ℓ-similarity inside the main algorithm. This contribution presents an alternative simple algorithm with running time $\mathcal{O}(n \log n)$, in which the computation is split into three phases. First, a compact representation of the gap table is created. Second, this representation is enriched with information about the length of a shortest word leading to the states. These two steps are independent of the parameter ℓ. Third, the ℓ-similarity is extracted by simple comparisons against ℓ. In particular, this approach allows the calculation of all the sizes of minimal ℓ-cover automata (for all valid ℓ) in the same time bound.

1 Introduction

Deterministic finite automata (DFA) are widely used in computer science due to their simplicity and flexibility. Their minimisation is one of the oldest problems that is motivated both theoretically and practically and almost every DFA toolkit implements it. More precisely, the DFA minimisation problem asks for a smallest DFA that recognises the same language as a given input DFA M. The asymptotically best solution is due to Hopcroft [9, 7], who presented an $\mathcal{O}(n \log n)$ algorithm where n is the number of states of M. Whether an asymptotically faster algorithm exists, remains one of the most challenging open questions in the area. In many applications the desired language L is finite. It was observed in [3] that membership of a word w in L can then be decided by: (i) checking

[⋆] This work was done when A. Maletti was visiting Wrocław University thanks to the support of the "Visiting Professors" programme of the Municipality of Wrocław

[⋆⋆] Supported by the MNiSW grant N206 492638 2010–2012 and by the Young Researcher scholarship of University of Wrocław.

[⋆⋆⋆] Supported by the German Research Foundation (DFG) grant MA/4959/1-1.

whether w is short (i.e., $|w| \leq \ell$ where $\ell = \max \{ |u| : u \in L\}$) and (ii) checking it with a DFA M. This allows M to accept words that are longer than ℓ, which yields that M need not recognise L. Thus, we arrive at the notion of 'cover automata'. We say that a DFA M is a *deterministic finite cover automaton* (DFCA or *cover automaton*) for a finite language L if $L(M) \cap \Sigma^{\leq \ell} = L$, where $\ell = \max\{ |u| : u \in L\}$ and $\Sigma^{\leq \ell}$ contains all words of length at most ℓ. It is a minimal DFCA for L if no DFCA for L has (strictly) fewer states.

It is well-known that the minimal DFCA for L can be substantially smaller than the minimal DFA for L. Already [3] presents a DFCA minimisation algorithm that runs in time $\mathcal{O}(n^2 \cdot \ell^2)$. It also allowed the input language to be presented as a DFA M, which could potentially recognise an infinite language. In that case, an explicit word length ℓ needs to be supplied. An ℓ-DFCA for M is simply a DFCA for $L(M) \cap \Sigma^{\leq \ell}$. Câmpeanu et al. [2] improved the minimisation algorithm for finite languages to $\mathcal{O}(n^2)$. Their algorithm can be trivially extended to arbitrary DFA, but it then runs in time $\mathcal{O}(n^2 \cdot \ell^2)$. The currently fastest algorithm for DFCA minimisation is due to Körner [12], who developed an algorithm that runs in time $\mathcal{O}(n \log n)$, and is a clever and refined modification of Hopcroft's algorithm for DFA minimisation.

Minimal DFCA are theoretically characterised [3, 12, 4]. All known algorithms for constructing a minimal ℓ-DFCA are based on a similarity relation $\sim_\ell$ on states, which is defined such that a minimal ℓ-DFCA consists of pairwise dissimilar states. The relation $\sim_\ell$ is defined using two very basic notions: (i) the level of a state, which is the length of a shortest word leading to it, and (ii) the gap between two states, which is the length of a shortest word on which they differ.

Lossy compression of DFA has received some attention recently, and DFCA minimisation can be considered as an instance. *Hyper-minimisation* [1] is another instance and aims to find a smallest DFA N for a given DFA M such that $L(M)$ and $L(N)$ have finite symmetric difference. This notion was refined to *ℓ-minimisation* [5], where the languages are allowed to differ only on words of length at most ℓ. Yet another variant was proposed by Schewe [13].

It is noteworthy that ℓ-minimisation and ℓ-DFCA minimisation are dual. It was already observed by Badr et al. [1] that there are languages L, which are best represented by a pair consisting of an ℓ-minimal automaton (that makes errors on words of length at most ℓ) and a minimal ℓ-DFCA. This combination can be substantially smaller than a single minimal DFA for L. An input word w is processed by such a pair by selecting the authorative DFA based on the word's length.

In principle, this approach works for all possible values of ℓ. Thus, it is desirable to construct an algorithm that decides for which value of ℓ the size of the representation is minimal. For this, we need to have algorithms that for a given DFA M return the size of an ℓ-minimal DFA and a minimal ℓ-DFCA for several values ℓ. We note that such an algorithm is known for ℓ-minimal DFA [6], and the current contribution adds the algorithm for minimal ℓ-DFCA.

In this paper, we give an alternative ℓ-DFCA minimisation algorithm, which proceeds in three phases. First, we calculate the function 'gap' and represent it compactly in a gap-tree. We show that its computation can be done by a slightly augmented version of HOPCROFT's algorithm, which means that it can be prepared in the DFA minimisation step. Second, we take the level of states into account and annotate the gap-tree. Up to this point, the computation is independent of the value of ℓ, and the obtained annotated gap-tree can be reused for all ℓ. In the third step, we identify the states that should be preserved in the minimal ℓ-DFCA (which naturally depends on ℓ) and determine its transition function. Our approach has several advantages. First, it is much easier to understand, verify, and implement. Its first phase closely resembles HOPCROFT's algorithm, which is well-known and understood. Second, since the first two phases are independent of ℓ, we can easily compute the size of all minimal ℓ-DFCA (for all valid ℓ) without overhead. In addition, we present an algorithm that constructs (a compact representation of) minimal ℓ-DFCA for consecutive values of ℓ in time $\mathcal{O}(n \log n)$.

We would like to point out that the minimisation algorithm presented in this paper shares the general outline with the ℓ-minimisation algorithm [6]: they both divide the computation of the minimal (with respect to the proper relation) DFA into phases, out of which only the last one depends on ℓ. Moreover, in both cases we present an ultrametric as an ultrametric tree and then annotate it. Due to differences in the similarity relations, the details vary significantly.

2 Preliminaries

In the following, let $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a minimal DFA, and let $m = |Q \times \Sigma|$ and $n = |Q|$. As usual, we let $\min \emptyset = \infty$. For every state $q \in Q$, we let $\text{level}(q) = \min \{\, |w| \,:\, \delta(q_0, w) = q\}$ and call it the level of q. Given two states $p, q \in Q$, we define their *gap* by

$$\text{gap}(p, q) = \min \{\, |w| \,:\, w \in L(p) \bigtriangleup L(q)\} \enspace ,$$

where $\bigtriangleup$ is the symmetric difference operator. Note that $d(p, q) = 2^{-\,\text{gap}(p,q)}$ with $2^{-\infty} = 0$ defines an ultrametric. We continue to work with $\text{gap}(p, q)$ because it is used in the ℓ-similarity relation $\sim_\ell$, which is defined by

$$p \sim_\ell q \iff \max(\text{level}(p), \text{level}(q)) + \text{gap}(p, q) > \ell \enspace ,$$

for all $p, q \in Q$. The currently fastest algorithm [12] for calculating minimal cover automata uses $\sim_\ell$, which in general is not an equivalence relation, but only a compatibility relation (i.e., reflexive and symmetric). Some additional, useful properties of $\sim_\ell$ are presented in [4]. In particular, they allow us to form an equivalence relation as follows.

Definition 1 (cf. [12, Definition 3]). *Let $\pi \colon Q \to P$ be a mapping for some $P \subseteq Q$ such that $\pi(p) = p$ for every $p \in P$. Then π is an* ℓ-similarity state decomposition *(ℓ-SSD) of Q if*

1. $\text{level}(q) \geq \text{level}(\pi(q))$ *for all $q \in Q$,*

2. $q \sim_\ell \pi(q)$ *for every* $q \in Q$*, and*
3. $p \not\sim_\ell p'$ *for all* $p, p' \in P$ *with* $p \neq p'$*.*

In other words, an ℓ-SSD is a partition of Q into $|P|$ blocks such that (1) each block has a representative with minimal level, (2) all elements in a block are ℓ-similar to their representative, and (3) the representatives of different blocks are pairwise ℓ-dissimilar. It is easy to observe that an ℓ-SSD $\pi\colon Q \to P$ contains a maximal (with respect to set inclusion) set P of pairwise ℓ-dissimilar states. Consequently, every ℓ-SSD π yields a minimal ℓ-DFCA by taking the quotient of M with respect to the equivalence relation π represents.

Theorem 2 (cf. [12, Theorem 1]). *For every ℓ-SSD $\pi\colon Q \to P$, the DFA $(M/\pi) = \langle P, \Sigma, \mu, \pi(q_0), F \cap P\rangle$ is a minimal ℓ-DFCA, where $\mu(p, a) = \pi(\delta(p, a))$ for every $p \in P$ and $a \in \Sigma$.*

Körner's algorithm constructs an ℓ-SSD using a clever modification of Hopcroft's algorithm [9]. It initially partitions the states into F and $Q \setminus F$ and then refines this partition while preserving Property 3 of Definition 1. Once the algorithm stops, also Property 2 of Definition 1 will be satisfied.

Part of the difficulty of Körner's algorithm stems from the fact that it takes both 'gap' and 'level' into account when refining the partition. Our approach separates these two properties. We show that $\text{gap}(p, q)$ can be calculated by a standard run of Hopcroft's algorithm. Moreover, the gap-matrix can be compactly represented as a gap-tree $\mathcal{G}$. With the help of $\mathcal{G}$, we can then compute an ℓ-SSD in a simpler manner by only taking 'level' into account.

3 Gap-Trees

The gap-matrix has size $\Theta(n^2)$, thus any algorithm that explicitly uses it is doomed to run in time $\Omega(n^2)$. To obtain a minimisation algorithm that runs in time $\mathcal{O}(m \log n)$ we need to represent it more compactly. This is achieved with the help of the gap-tree $\mathcal{G}$, which contains a leaf for each state of Q. The tree is organised such that each subtree contains only states whose pairwise gap exceeds a certain value. More precisely, for each subtree t' there exists $s \in \mathbb{N}$ such that

$$\text{gap}(p, q) \begin{cases} \geq s & \text{if } p \text{ and } q \text{ occur in } t' \\ < s & \text{otherwise.} \end{cases}$$

In the next section, it is shown that gap tree can be created during a standard run of a slightly augmented variant of Hopcroft's algorithm.

Before we start with the formal definition, we recall some notions on trees. We generally use rooted trees, which are special undirected graphs with a dedicated vertex r (the root) such that there is exactly one path from each vertex to r. Moreover, we use weighted edges, where the edge weights are nonnegative integers. The sum of the edge weights along the unique path from a vertex v to the root r is denoted by $\text{d}(v)$ and called the *depth of* v. A leaf is a vertex with

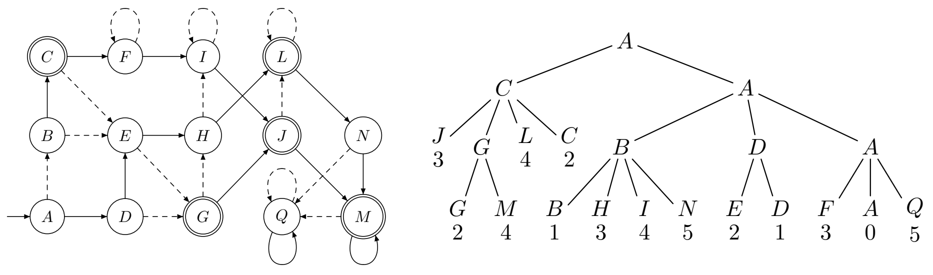

Fig. 1. Example DFA (left) and a gap-tree (right) for it.

only one adjacent edge. A tree is an *ultrametric tree* [8, 10, 11] if the depth of all leaves is equal. Finally, for two vertices v and v', their join $v \vee v'$ is the lowest common ancestor (i.e., the deepest vertex such that both v and v' occur in its subtree).

Definition 3. *An ultrametric tree for* gap *(for short:* gap tree*) is an ultrametric tree with leaves Q such that* $\text{gap}(p, q) = \text{d}(p \vee q)$ *for all $p, q \in Q$ with $p \neq q$.*

Next, we want to determine representatives of similarity blocks. Since each vertex of the gap-tree determines a subtree and thus a block of states, which are the states that occur in the subtree, we assign a state to each vertex. To satisfy Property 1 of Definition 1, we select a state with minimal level among all states assigned to the direct subtrees. Formally, given a gap-tree $\mathcal{G}$ with vertices V, we let state: $V \to Q$ be a mapping such that (i) $\text{state}(q) = q$ for all $q \in Q$, (ii) $\text{state}(v) = \text{state}(v')$ for all $v \in V \setminus Q$, where v' is some direct child vertex of v, and (iii) $\text{level}(\text{state}(v)) \leq \text{level}(\text{state}(v'))$ for all $v \in V$ and v' being a direct child vertex of v. Note there can be several mappings 'state' that fulfill the requirements (i)–(iii), which correspond to different choices of representatives. In the following, we assume that 'state' is any such mapping.

The selected mapping 'state' labels all vertices of $\mathcal{G}$ with a state of Q. Recall that $\text{state}(q) = q$ for every $q \in Q$. Consequently, for every $q \in Q$ there exists a minimal (i.e., of minimal depth) vertex $v_q \neq q$ such that $\text{state}(v) = q$ for all vertices besides v_q along the path (towards the root) starting in the leaf q to v_q. Note that the vertex v_q is unique, and called the *termination vertex* of q. The *termination state* of $q \in Q$ is $\text{state}(v_q)$. Recall that r is the root vertex. Note that the termination state of q is always different from q unless $q = \text{state}(r)$. Moreover, for all states $q \neq \text{state}(r)$ we have $\text{gap}(q, \text{state}(v_q)) = \text{d}(q \vee \text{state}(v_q)) = \text{d}(v_q)$, which motivates the following definitions.

Definition 4. *For every $q \in Q$, let*

– *the state-gap $g(q)$ be such that*

$$g(q) = \begin{cases} -\infty & \textit{if } q = \text{state}(r), \\ \text{d}(v_q) & \textit{otherwise.} \end{cases}$$

– $\text{value}(q) = \text{level}(q) + g(q)$.

Example 5. Let us consider the minimal DFA and the gap-tree for it that are displayed in Fig. 1. Below the leaves of the gap-tree we annotated the state's level. In addition, we already labelled the inner nodes with states. Now, we can determine the termination state for each state. For example, C is the termination state of G because it is the first label on the path from the leaf G towards the root that differs from G. Consequently, $g(G) = \text{d}(v_G) = 1$ and $\text{value}(G) = 2 + 1 = 3$. Overall, we obtain:

$$\begin{array}{lllll} \text{value}(A) = -\infty & \text{value}(B) = 2 & \text{value}(C) = 2 & \text{value}(D) = 2 & \text{value}(E) = 4 \\ \text{value}(F) = 5 & \text{value}(G) = 3 & \text{value}(H) = 5 & \text{value}(I) = 6 & \text{value}(J) = 4 \\ \text{value}(L) = 5 & \text{value}(M) = 6 & \text{value}(N) = 7 & \text{value}(Q) = 7. & \end{array}$$

It is important to note that all the previous notions on the gap-tree are independent of the selection of ℓ. Nevertheless, we can use them to transform the gap-tree $\mathcal{G}$ into an ℓ-SSD. Let $P = \{\, q \in Q : \text{value}(q) \leq \ell\}$. Note that $P \neq \emptyset$ because $\text{state}(r) \in P$. For every state $q \in Q$, its *ℓ-state* $\pi(q)$ is the label $\text{state}(v)$ of the first vertex v on the path from q to the root r such that $\text{value}(\text{state}(v)) \leq \ell$.

Lemma 6. *The ℓ-state mapping π is an ℓ-SSD.*

Proof. We have to show the conditions of Definition 1. For every $p \in P$ we have $\text{value}(p) \leq \ell$. Consequently, their ℓ-state $\pi(p)$ is p. Moreover, for every $q \in Q$ we have $\text{level}(q) \geq \text{level}(\pi(q))$ because $\pi(q)$ is the label of an ancestor of q. Let us continue with Condition 2 of Definition 1. It trivially holds for $q = \pi(q)$, so suppose that $q \in Q$ is such that $q \neq \pi(q)$. Consequently, $q \notin P$. Let p be the label of the last vertex v on the path from q to the root r such that $\text{value}(\text{state}(v)) > \ell$. Clearly, the next vertex along this path is the ℓ-vertex of q, which is labelled $\pi(q)$. Note that $p = q$ is possible. Then $q \vee \pi(q) = p \vee \pi(q)$ and thus $\text{gap}(q, \pi(q)) = g(p)$. In addition, $\text{level}(q) \geq \text{level}(p) \geq \text{level}(\pi(q))$. These two estimations together yield that

$$\begin{aligned} &\max(\text{level}(q), \text{level}(\pi(q)) + \text{gap}(q, \pi(q)) \\ &= \text{level}(q) + g(p) \geq \text{level}(p) + g(p) = \text{value}(p) > \ell \ , \end{aligned}$$

which proves $q \sim_\ell \pi(q)$.

Finally, we have to show Condition 3 of Definition 1. Let $p, p' \in P$ be such that $p \neq p'$. Consequently, $\text{value}(p) \leq \ell$ and $\text{value}(p') \leq \ell$. Without loss of generality, suppose that (i) $\text{level}(p) \geq \text{level}(p')$, (ii) $p \vee p'$ is not labelled with p. If these conditions are not met for the pair (p, p'), then they are met for the pair (p', p). Since $\text{state}(p \vee p') \neq p$, the termination vertex of p is on the path from p to $p \vee p'$ and so $g(p) \geq \text{d}(p \vee p') = \text{gap}(p, p')$. Taking this and assumption (i) into account, we obtain

$$\begin{aligned} &\max(\text{level}(p), \text{level}(p')) + \text{gap}(p, p') \\ &\leq \text{level}(p) + g(p) = \text{value}(p) \leq \ell \ , \end{aligned}$$

which proves $p \not\sim_\ell p'$. □

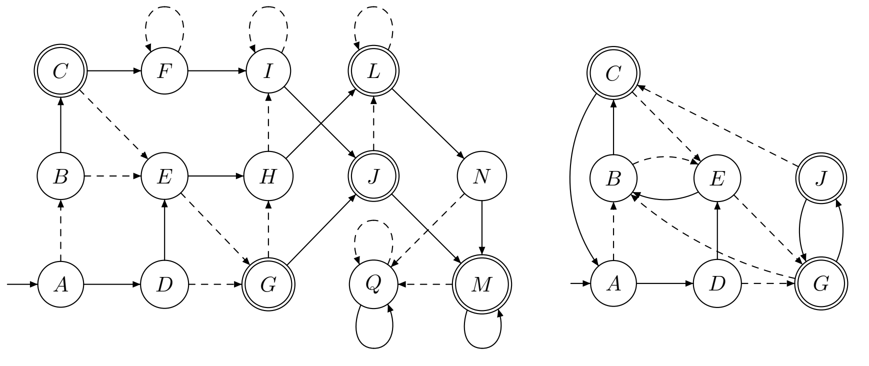

Fig. 2. Example DFA (left) and a minimal 4-DFCA (right) for it.

Example 7 (Example 5 continued). Take $\ell = 4$, then $P = \{A, B, C, D, E, G, J\}$. Besides the obvious entries (identity on P) we have:

$$\pi(F) = \pi(Q) = A \qquad \pi(H) = \pi(I) = \pi(N) = B \qquad \pi(L) = C \qquad \pi(M) = G \ .$$

The resulting minimal 4-DFCA is displayed in Fig. 2.

Lemma 6 shows that the gap-tree with the help of 'value' indeed contains a characterisation of $\sim_\ell$ for all potential ℓ. This allows a fast construction of a minimal ℓ-DFCA for M whenever a gap-tree $\mathcal{G}$ is provided. We say that a gap-tree $\mathcal{G}$ with vertices V is *small* if $|V| \in \mathcal{O}(n)$.

Theorem 8. *Given a small gap-tree $\mathcal{G}$ with $\mathrm{d}(q) = s$ for all $q \in Q$, we can*

1. *calculate the sizes of all minimal ℓ-DFCA (for all valid ℓ) in time $\mathcal{O}(m+s)$,*
2. *construct a minimal ℓ-DFCA for a given ℓ in time $\mathcal{O}(m)$, and*
3. *iteratively construct (representations of) minimal ℓ-DFCA for all ℓ in time $\mathcal{O}(m \log n + s)$*

Proof. Let V be the set of vertices of $\mathcal{G}$. First, we compute a proper state labelling state: $V \to Q$ in the obvious manner. This can be done in time $\mathcal{O}(n)$ because $\mathcal{G}$ is small. Similarly, we can compute 'level' in time $\mathcal{O}(m)$ because every transition needs to be considered only once. A simple bottom-up procedure on $\mathcal{G}$ can calculate value(q) for every state q using 'level' and 'state'. Overall, we can complete these steps in time $\mathcal{O}(m)$.

For the first claim, we sort the elements of Q by their 'value' in time $\mathcal{O}(n+s)$ using, for example, Counting-Sort. We know that value$(q) \leq n + s$ for every $q \in Q$, hence we can obtain the mentioned time-bound for sorting. From this sorted list of states, we can now determine the sizes of all minimal ℓ-DFCA in time $\mathcal{O}(n+s)$ by iteration over the list because for a given ℓ the size of a minimal ℓ-DFCA is $|\{\, q \,:\, \text{value}(q) \leq \ell \}|$.

Next, let us move to the second claim. Theorem 2 and Lemma 6 show that given an efficient representation of an ℓ-state mapping π, we can construct a minimal ℓ-DFCA in time $\mathcal{O}(m)$. Consequently, it only remains to determine an ℓ-state mapping $\pi \colon Q \to P$. Clearly, the set $P = \{ q : \text{value}(q) \leq \ell \}$ can be constructed in time $\mathcal{O}(n)$ by a simple iteration over Q. Finally, we need to determine π. To this end, we traverse the gap-tree $\mathcal{G}$ top-down. Every time, we encounter a vertex v' such that $\text{value}(\text{state}(v')) > \ell$ but $\text{value}(\text{state}(v)) \leq \ell$, where v is the direct ancestor of v', we set $\pi(q) = \text{state}(v)$ for all states $q \in Q$ that occur in the subtree of v'. Overall, this can be achieved in time $\mathcal{O}(|V|)$. Since $\mathcal{G}$ is small, we obtain the time bound $\mathcal{O}(m)$.

Finally, we have to show how to create minimal ℓ-DFCA sequentially, so that the total execution time is $\mathcal{O}(m \log n + s)$. Let $\ell_{\max+1} = \max_{q \in Q} \text{value}(q)$. In each step $\ell \in \{\ell_{\max}, \dots, 1, 0\}$ our algorithm keeps the states $P_\ell = \{ q : \text{value}(q) \leq \ell \}$. Consequently, it merges each state $q \in P_{\ell+1}$ such that $\text{value}(q) = \ell + 1$ into its terminating state p, which by construction satisfies

$$\text{value}(p) = \text{level}(p) + g(p) \leq \text{level}(q) + (g(q) - 1) = \text{value}(q) - 1 \leq \ell \enspace .$$

However, to obtain the stated running time, we need to organise the process properly. First, we note that there is a change in at most n steps because there can be at most n merges. Thus, we first filter out the steps, in which no changes occur. This can be done in time $\mathcal{O}(s)$. Second, we represent the DFA as a list of transitions. For each state q, we keep a list of all pairs (a, p) such that $\delta(q, a) = p$, where p is implemented as a pointer to a pointer to the actual state p, which allows a fast modification of all transitions leading to p by simply replacing the final pointer to p. In addition, for every state q, we keep a counter $c(q)$, which is initially 1 and counts how many states were merged into q. Now assume that we want to merge the state q into p. First, we assume that $c(q) \leq c(p)$. In this case, we simply redirect each incoming transition of q to p (by a constant-time pointer replacement). However, if $c(q) > c(p)$, then we redirect each incoming transition of p to q (i.e., we do not merge q into p, but rather merge p into q). In addition, we replace the outgoing transitions of q by the outgoing transitions of p, which can be done in constant time by simply replacing the pointer to the list. We complete this case by renaming q to p. Finally, in both cases we update $c(p)$ by $c(p) \leftarrow c(p) + c(q)$. In this manner, every time the transition target $\delta(q, a)$ is modified due to a merge, the value $c(\delta(q, a))$ at least doubles. Since $c(q) \leq n$ for each $q \in Q$, each transition can be modified at most $\log n$ times. Consequently, we obtain the overall running time $\mathcal{O}(m \log n + s)$. □

Note that the third statement of Theorem 8 only provides a (compact) representation of the minimal ℓ-DFCA in the presented running time $\mathcal{O}(m \log n + s)$. If we output the obtained DFCA for all ℓ, then we require time $\mathcal{O}(m^2 \log n)$ because we need $\mathcal{O}(m)$ steps for each output DFCA. The summand s disappears due to the fact that it can always be chosen such that $s \leq n^2 \leq m^2$.

4 Computing a gap-tree

We already showed that we can easily construct minimal ℓ-DFCA provided that we have access to a small gap-tree $\mathcal{G}$ for M. In this section, we show how to

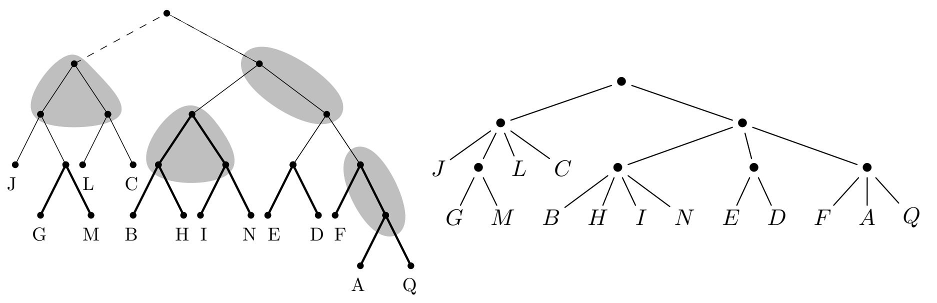

Fig. 3. The pre-gap tree (left) for the DFA of Fig. 1. The gap-tree (right) of Fig. 1 can be obtained from it by merging appropriate nodes, which are marked in grey. The edges labelled with different 'gap' were drawn in different styles (dashed = 0, normal = 1, thick = 2).

construct such a $\mathcal{G}$. Actually, a simple modification of HOPCROFT's algorithm [9] can perform the construction for us. Roughly speaking, we keep track of the length of words that cause a split of a set of states in the run of the algorithm. Already KÖRNER's algorithm [12] followed a similar strategy. Our modification is less drastic and yields a solution that is simpler and easier to understand.

Algorithm 1 presents the slightly modified version of HOPCROFT's algorithm that is suitable for our purposes. It creates a *pre-gap tree*, which keeps track of how the final partition was obtained. In particular, it stores the lengths of the used splitting words. The length of such a splitting word coincides with the gap between the affected states. The obtained pre-gap tree (see Fig. 3) is basically a binarisation of a gap-tree. It can easily be transformed into a gap-tree by merging appropriate nodes.

We marked the modifications (compared to a standard implementation of HOPCROFT's algorithm) by $\triangleright$. Clearly, all lines referring to gap calculations are new. However, they do neither affect the correctness of the overall algorithm nor the analysis of its run-time. In addition, the queue T is restricted to a FIFO-queue, which is essential for our purposes. Finally, although we split Q' into Q_{r-1} and Q_r in line 11, we do not replace Q' in T. When $v_{Q'}$ is extracted from T, we no longer have Q' as an element of P. However, we can recreate it by listing all the states that occur in the subtree of $v_{Q'}$. A similar approach was also used by KÖRNER [12], who proved that this does not affect the running time.

Next, we show that the pre-gap tree has the following properties:

1. It is a binary tree.
2. The states p and q point to the same node if and only if $p = q$.
3. If p and q with $p \neq q$ point, respectively, to v_p and v_q, then the edges to $v_p \vee v_q$ are labelled with $\text{gap}(p, q)$.

The first statement follows clearly from HOPCROFT's strategy. Every time a leaf turns into an inner node in line 13, two children are created. Moreover,

Algorithm 1 Modification of HOPCROFT's algorithm

```
1: $Q_1 \leftarrow F$, $Q_2 \leftarrow Q \setminus F$, $r \leftarrow 2$, $P \leftarrow \{Q \setminus F, F\}$
2: $T \leftarrow \{(v_F, 1)\}$                                                    ▷ FIFO queue
3: create $v_Q$ and its children $v_F, v_{Q\setminus F}$, gap$(v_F, v_Q) \leftarrow$ gap$(v_{Q\setminus F}, v_Q) \leftarrow 0$   ▷
4: put two-way pointers $v_F \leftrightarrow F$ and $v_{Q\setminus F} \leftrightarrow Q \setminus F$        ▷
5: while $T \neq \emptyset$ do
6:     $(v_{Q_i}, k_i) \leftarrow$ first from $T$
7:     for $a \in \Sigma$ do
8:         $Q_a = \{ q \,:\, \delta(q,a) \in Q_i\}$
9:         for $Q' \in P$ such that $Q' \not\subseteq Q_a$ and $Q' \cap Q_a \neq \emptyset$ do
10:            $r \leftarrow r + 2$
11:            $Q_{r-1} \leftarrow Q' \cap Q_a$, $Q_r \leftarrow Q' \setminus Q_a$
12:            $P \leftarrow (P \setminus \{Q'\}) \cup \{Q_{r-1}, Q_r\}$
13:            create nodes $v_{Q_{r-1}}$, $v_{Q_r}$ and edges $(v_{Q_{r-1}}, v_{Q'})$, $(v_{Q_r}, v_{Q'})$   ▷
14:            gap$(v_{Q_{r-1}}, v_{Q'}) \leftarrow$ gap$(v_{Q_r}, v_{Q'}) \leftarrow k_i$                   ▷
15:            if $|Q_{r-1}| > |Q_r|$ then
16:                add $(v_{Q_r}, k_i + 1)$ to $T$                ▷ Do not remove $Q'$ from $T$
17:            else
18:                add $(v_{Q_{r-1}}, k_i + 1)$ to $T$
19:     for $Q' \in P$ do
20:         for $q \in Q'$ do
21:             add pointer from $q$ to $v$, where $Q'$ points to $v$      ▷ Partition of states
```

the previous line removed the corresponding set from P, which yields that the vertex is never split again. In particular, this statement yields that the pre-gap tree is small. The second statement is the correctness of HOPCROFT's algorithm, so we do not reprove it. Before, we can prove the third (and essential) statement, we first identify some properties of the maintained data structure.

Lemma 9 (cf. [12, Lemma 4]). *Let $(v_{Q_1}, k_1), \ldots, (v_{Q_s}, k_s)$ be the complete sequence of elements added to T during the run of Algorithm 1. Then $k_{i-1} \leq k_i$ for all $i \in \{2, \ldots, s\}$.*

Proof. We prove the statement by induction. For $i = 2$ it is obvious because $k_1 = 1$ and $k_2 = 2$. Now, let $i \geq 3$. The element (v_{Q_i}, k_i) was put into T while processing (v_{Q_j}, k_j) for some $j \leq i - 1$. Due to the FIFO strategy, its predecessor $(v_{Q_{i-1}}, k_{i-1})$ was put into T while processing $(v_{Q_{j'}}, k_{j'})$ for some $j' \leq j$. By the induction assumption, we have $k_{j'} \leq k_j$. Consequently, we obtain that $k_{i-1} = k_{j'} + 1 \leq k_j + 1 = k_i$. □

Lemma 10. *For any two inequivalent states $p, q \in Q$ there is a set $Q' \in P$ at some point during the execution of Algorithm 1 that is split into Q_{r-1} and Q_r with $p \in Q_{r-1}$ and $q \in Q_r$. The corresponding edges $(v_{Q_{r-1}}, v_{Q'})$ and $(v_{Q_r}, v_{Q'})$ are labelled by* gap(p, q).

Proof. Since p and q are inequivalent, the states p and q will be split. Thus, the set Q' with the given properties exists. It remains to prove the property about the gap. Let $\text{gap}'(p,q) = \text{gap}(v_{Q_{r-1}}, v_{Q'})$. Next, we show that $\text{gap}(p,q) = \text{gap}'(p,q)$. To this end, we first show that $\text{gap}(p,q) \leq \text{gap}'(p,q)$ and then demonstrate that $\text{gap}(p,q) \geq \text{gap}'(p,q)$, which will conclude the proof.

The first part is shown by induction on the number i of elements of T considered by the algorithm. If $i = 0$, then $\text{gap}(p,q) = 0$ because exactly one of $\{p,q\}$ is in F. Since line 3 assigns the same gap, the claim holds. The inequivalent states p and q are eventually split by the algorithm. Let $Q' \in P$ be the element such that $\{p,q\} \subseteq Q'$ before they are split. Intuitively, the element $(v_{Q''}, k)$ of T that caused the split has the property that there exists a letter $a \in \Sigma$ such that exactly one of the states $p_a = \delta(p,a)$ and $q_a = \delta(q,a)$ is in Q''. Consequently, $p_a \neq q_a$ and $\text{gap}(p_a, q_a) \leq \text{gap}'(p_a, q_a)$ by the induction hypothesis (because p_a and q_a must have been split in a previous iteration). Then

$$\text{gap}(p,q) \leq \text{gap}(p_a, q_a) + 1 \leq \text{gap}'(p_a, q_a) + 1 = k = \text{gap}'(p,q) \ ,$$

which proves the induction step.

Finally, we show that $\text{gap}'(p,q) \leq \text{gap}(p,q)$ for all pairs (p,q) of states with $p \neq q$. Let $w = a_1 \cdots a_m$ be the shortest string such that exactly one of the states $\delta(p,w)$ and $\delta(q,w)$ is in F. Moreover, let (i) $p_0 = p$ and $q_0 = q$, and (ii) $p_i = \delta(p_{i-1}, a_i)$ and $q_i = \delta(q_{i-1}, a_i)$ for every $i \in \{1, \ldots, m\}$. Let us consider the maximal i such that $\text{gap}(p_i, q_i) < \text{gap}'(p_i, q_i)$. Trivially, we have $i < m$ because $\text{gap}(p_m, q_m) = \text{gap}'(p_m, q_m) = 0$, which follows because exactly one of $\{p_m, q_m\}$ is in F. By the first statement and the maximality of i, we have $\text{gap}(p_{i+1}, q_{i+1}) = \text{gap}'(p_{i+1}, q_{i+1})$. Due to the algorithm, there exists an element $(v_{Q''}, k)$ of T and $a \in \Sigma$ such that $\text{gap}'(p_i, q_i) = k$, where $p' = \delta(p_i, a)$, $q' = \delta(q_i, a)$, and exactly one of $\{p', q'\}$ is in S. The latest the split can happen is due to (p_{i+1}, q_{i+1}), but it can happen earlier, which allows us to conclude by Lemma 9 that

$$\begin{aligned}\text{gap}'(p_i, q_i) &= k \leq \text{gap}'(p_{i+1}, q_{i+1}) + 1 = \text{gap}(p_{i+1}, q_{i+1}) + 1 \\ &= \text{gap}(p_i, q_i) \ ,\end{aligned}$$

where the last equality follows from the fact that w is the shortest word. Consequently, $\text{gap}'(p_i, q_i) \leq \text{gap}(p_i, q_i)$, which contradicts the assumption and completes the proof. □

Now Property 3 of the pre-gap tree is an easy corollary of Lemma 10: consider any two inequivalent states p and q. The set Q' from Lemma 10 corresponds to the node $v_p \vee v_q$ in the pre-gap tree. Furthermore the lemma asserts that the edges to $v_p \vee v_q$ are labelled by $\text{gap}(p,q)$.

To obtain a gap-tree $\mathcal{G}$ from the pre-gap tree for the DFA M, it is enough to merge connected parts of the pre-gap tree with incoming edges labelled with the same value k into a single vertex v such that $\text{d}(v) = k$ (see Fig. 3). Moreover, Lemma 10 shows that $\text{d}(q) \leq n$ for every $q \in Q$, which allows us to state our main theorem.

Theorem 11. *For all DFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ with $m = |Q \times \Sigma|$ and $n = |Q|$, we can perform the following in time $\mathcal{O}(m \log n)$:*

1. *Calculate the sizes of all minimal ℓ-DFCA (for all valid ℓ).*
2. *Construct a minimal ℓ-DFCA for a given ℓ.*
3. *Iteratively construct (representations of) minimal ℓ-DFCA for all ℓ.*

References

1. Badr, A., Geffert, V., Shipman, I.: Hyper-minimizing minimized deterministic finite state automata. RAIRO Theoret. Inform. Appl. 43(1), 69–94 (2009)
2. Câmpeanu, C., Paun, A., Yu, S.: An efficient algorithm for constructing minimal cover automata for finite languages. Int. J. Found. Comput. Sci. 13(1), 83–97 (2002)
3. Câmpeanu, C., Santean, N., Yu, S.: Minimal cover-automata for finite languages. Theor. Comput. Sci. 267(1–2), 3–16 (2001)
4. Champarnaud, J.M., Guingne, F., Hansel, G.: Similarity relations and cover automata. RAIRO Theoret. Inform. Appl. 39(1), 115–123 (2005)
5. Gawrychowski, P., Jeż, A.: Hyper-minimisation made efficient. In: Proc. 34th Int. Symp. Mathematical Foundations of Computer Science. LNCS, vol. 5734, pp. 356–368. Springer (2009)
6. Gawrychowski, P., Jeż, A., Maletti, A.: On minimising automata with errors. Corr `abs/1102.5682` (2011)
7. Gries, D.: Describing an algorithm by Hopcroft. Acta Inf. 2(2), 97–109 (1973)
8. Hartigan, J.A.: Representation of similarity matrices by trees. J. Amer. Statist. Assoc. 62(320), 1140–1158 (1967)
9. Hopcroft, J.E.: An $n \log n$ algorithm for minimizing states in a finite automaton. In: Kohavi, Z. (ed.) Theory of Machines and Computations, pp. 189–196. Academic Press (1971)
10. Jardine, C.J., Jardine, N., Sibson, R.: The structure and construction of taxonomic hierarchies. Math. Biosci. 1(2), 173–179 (1967)
11. Johnson, S.C.: Hierarchical clustering schemes. Psychometrika 32(3), 241–254 (1967)
12. Körner, H.: A time and space efficient algorithm for minimizing cover automata for finite languages. Int. J. Found. Comput. Sci. 14(6), 1071–1086 (2003)
13. Schewe, S.: Beyond hyper-minimisation — Minimising DBAs and DPAs is NP-complete. In: Proc. Ann. Conf. Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 8, pp. 400–411. Schloss Dagstuhl (2010)