Applied Intelligence (2022) 52:3029–3050
https://doi.org/10.1007/s10489-021-02488-y

An adaptive parallel algorithm for finite language decomposition

Tomasz Jastrząb[1] · Zbigniew J. Czech[1] · Wojciech Wieczorek[2]

Accepted: 29 April 2021 / Published online: 26 June 2021

Abstract
The computationally hard problem of finite language decomposition is investigated. A finite language L is decomposable if there are two languages L_1 and L_2 such that $L = L_1L_2$. Otherwise, L is prime. The main contribution of the paper is an adaptive parallel algorithm for finding all decompositions L_1L_2 of L. The algorithm is based on an exhaustive search and incorporates several original methods for pruning the search space. Moreover, the algorithm is adaptive since it changes its behavior based on the runtime acquired data related to its performance. Comprehensive computational experiments on more than 4000 benchmark languages generated over alphabets of various sizes have been carried out. The experiments showed that by using the power of parallel computing the decompositions of languages containing more than 200000 words can be found. Decompositions of languages of that size have not been reported in the literature so far.

Keywords Adaptive parallel algorithm · Parallel exhaustive search with pruning · Finite language decomposition · Primality test · Formal language theory

1 Introduction

A finite language L is decomposable (or composite) if there are two nontrivial languages L_1 and L_2 such that $L = L_1L_2$. Otherwise, L is prime. It was proved that the complexity of deciding primality of a finite language is NP-hard [1].

The main contribution of this paper is an adaptive parallel algorithm that finds all decompositions L_1L_2 of L, or concludes that L is prime when no decomposition is found. The algorithm is based on an exhaustive search, and incorporates several original methods for pruning the search space. Moreover, the algorithm is adaptive; it changes its behavior based on the runtime acquired data related to the performance of its recursive phase. Since the question of decomposing finite languages is computationally hard, a motivation of our work is to investigate to what extent the power of parallel computing makes it possible to tackle that question for large-size instances of finite languages.

The decomposition algorithm has a number of applications. It is useful for determining the prime decomposition of a supercode. A finite language L can be represented by the list of words. However, such representation is memory intensive if L is large. Therefore a better option is to decompose L and store its factors L_1 and L_2. The decomposition algorithm can be used to find a context-free grammar G. Given the sets of words S_+ and S_-, called examples and counterexamples, a grammar G should be found that accepts words in set S_+, and rejects words in set S_-. We describe these applications in more detail in Examples 3–6.

The rest of the paper is organized into six sections. Section 2 presents a survey of previous work on finite language decomposition. Section 3 recalls selected concepts of the theory of languages and automata. Section 4 describes the basic algorithm, which has been a starting point for developing the adaptive algorithm proposed in Section 5. Section 6 reports on the results of the computational experiments conducted using the algorithms. Section 7 contains conclusions and future work.

2 Related work

M. Martin and T. Kutsia [2] proposed a representation of regular languages by linear systems of language equations, which is suitable for computing left and right factors of a regular language. An n-subfactorization of a regular

✉ Tomasz Jastrząb
Tomasz.Jastrzab@polsl.pl

1 Department of Algorithmics and Software, Silesian University of Technology, Gliwice, Poland

2 Department of Computer Science and Automatics, University of Bielsko-Biała, Bielsko-Biała, Poland

language L is a tuple of languages $(L_1, L_2, \ldots, L_n)$ for some $n \geq 1$ such that the language concatenation $L_1L_2 \ldots L_n$ is a proper subset of L, so $L_1L_2 \ldots L_n \subsetneq L$. A left (resp. right) factor of L is the leftmost (resp. rightmost) term in a factorization of L. The algorithm for computing the sets of left and right factors of L is proposed.

S. Afonin and D. Golomazov [3] presented an algorithm for constructing a minimal union-free decomposition of a regular language L. A representation $L = L_1 \cup L_2 \cup \ldots \cup L_k$ is called a union-free decomposition of L iff L_i is a union-free language for all $i = 1, 2, \ldots, k$. The decomposition is called minimal iff there is no other union-free decomposition of L with fewer factors. The algorithm for constructing a minimal union-free decomposition for a given regular language L is provided. The algorithm involves an exhaustive search, so its computational complexity is exponential in the size of the DFA accepting L.

W. Wieczorek and A. Nowakowski [4] considered the problem of finding a multi-decomposition of a finite language $W = \{w_1, w_2, \ldots, w_n\}$, $n \geq 1$, that contains words w_i of fixed length $d \geq 2$, composed of symbols taken from the nonempty finite alphabet Σ. It is assumed that a desired multi-decomposition is a set of concatenations L_iR_i whose union contains W, that is $W \subseteq L_1R_1 \cup L_2R_2 \cup \ldots \cup L_mR_m$ for some $m \geq 1$ depending on the size of W. The nonempty sets L_i and R_i, which include the prefixes and suffixes of words $w_i \in W$, respectively, are the subsets of complete sets of prefixes and suffixes obtained from all possible splits of words $w_i \in W$. Denote the splits by $w_i = u_{ij}v_{ij}$, $i \in [1, n]$, and $j \in [1, d-1]$. The prefix u_{ij} consists of j leading symbols of w_i, while the suffix v_{ij} consists of the $d - j$ trailing symbols of w_i. A multi-decomposition is related to the cliques of an undirected graph $G = (V, E)$ with the set of vertices $V = \{(u_{ij}, v_{ij}) | w_i = u_{ij}v_{ij}, w_i \in W\}$ and the set of edges $E = \{((u_{ij}, v_{ij}), (u_{kl}, v_{kl})) | u_{ij}v_{kl}, u_{kl}v_{ij} \in W\}$. It can be shown that each concatenation L_iR_i of a multi-decomposition is represented by the corresponding clique in graph G, that is $\{(u_{t_1}, v_{t_1}), (u_{t_2}, v_{t_2}), \ldots, (u_{t_r}, v_{t_r})\}$ where $L_i = \cup_{j=1}^{r}\{u_{t_j}\}$ and $R_i = \cup_{j=1}^{r}\{v_{t_j}\}$. The authors provide a randomized algorithm to find all cliques in graph G, which runs in polynomial time with respect to the size of W. As an example, consider language $W = \{aa, ab\}$. The graph G for it has two vertices labeled (a, a) and (a, b) ($u_{11} = v_{11} = a$, $u_{21} = a$, $v_{21} = b$), and one edge $((a, a), (a, b))$, as $u_{11}v_{21} = ab$ and $u_{21}v_{11} = aa$ are in W. There is only one clique in the graph, so $L_1 = u_{11} \cup u_{21} = \{a\}$ and $R_1 = v_{11} \cup v_{21} = \{a, b\}$, and finally $W = \{a\}\{a, b\}$. The proposed method of finding a multi-decomposition was applied to create an opening book for the Toads-and-Frogs game.

H. J. Bravo et al. [5] investigate the manufacturing problem that involves batches of identical or similar items produced together for different sized production runs. A production run, consisting of a fixed number of interleaved workcycles, and is controlled by a monolithic supervisor, whose operation is described using a deterministic finite automaton (DFA). The automaton accepts a regular language R over some finite alphabet Σ, with states representing points in time, and transitions representing events occurring within a production run. Events are operations that are performed by machines (for example, take a workpiece, process it, etc.) to obtain the product. An event a has a specific time of execution $d(a)$, and belongs to the alphabet of a DFA, $a \in \Sigma$. A production run, being a sequence $s = a_1a_2 \ldots a_n$ of events, has a makespan equal to the sum of the event execution times, $\sum_{i=1}^{n} d(a_i)$. For a product there can be a number of sequences s that differ in the order of individual events a_i, with each sequence corresponding to a route in a DFA. The paper proposes a factorization-based approach to compute a route with the minimum makespan. As for a given product P the production run length is finite (measured by the number of events), the finite language L is extracted from a regular language R, $L \subseteq R$, which describes all possible routes for P. Then L is decomposed into factors $L = L_1L_2 \ldots L_k$ for some k, where each factor L_i describes a set of possible subroutes betwen the so called symmetrically reachable idle states (the notion introduced in the paper). Finally, for all sets L_i, $i \in [1, k]$, the subroutes r_i of the minimum makespan are determined, which give the optimal route $r = r_1r_2 \ldots r_k$. The authors claim that the proposed approach mitigates the computational complexity of finding route r.

Several decomposition algorithms were developed for codes, which are sets of words and hence they are formal languages. Codes are categorized by their defining properties, for example, prefix-freeness, suffix-freeness, infix-freeness, etc.

Y. -S. Han and K. Salomaa [6] studied solid codes defined as follows: a set S of words is a solid code, if S satisfies two conditions: (i) no word of S is a subword of another word of S (infix-freeness), and (ii) no prefix of a word in S is a suffix of a word in S (overlap-freeness). In other words, S has to be an infix code and all words of S should not overlap. Moreover, a language L is a regular solid code if L is regular and a solid code. The paper proposed two algorithms related to the decomposition problem for solid codes. The first algorithm determines in polynomial time whether or not a given regular language is a solid code. The second efficient algorithm finds a prime solid code decomposition for a regular solid code when it is not prime.

K. V. Hung [7] considered the prime decomposition of supercodes. A finite language L is a supercode, if no word in L is a proper permu-subword of another word in it. Let u

and v be words of L. A word u is called a permu-subword of v, if u is a subword of v in which symbol permutations are allowed. A supercode L is prime if $L \neq L_1L_2$ for any supercodes L_1 and L_2. A linear-time algorithm was provided that for a given supercode L discovered that it was prime, or returned the unique sequence of prime supercodes $L_1, L_2, \ldots, L_{k+1}$ such that $L = L_1L_2 \ldots L_{k+1}$ with $k \geq 1$.

In addition to the above recent work on supercodes, there were previous works on this subject. J. Czyzowicz et al. [8] proved that for a given prefix-free regular language L, the prime prefix-free decomposition is unique, and the decomposition for L that is not prime can be computed in $O(m)$ worst-case time, where m is the size of the minimal DFA accepting L. Y. -S. Han et al. [9] investigated the prime infix-free decomposition of infix-free regular languages and demonstrated that the prime infix-free decomposition is not unique. An algorithm for the infix-free primality test of an infix-free regular language was given. It was also shown that the prime infix-free decomposition can be computed in polynomial time. Y. -S. Han and D. Wood [10] investigated the finite outfix-free regular languages. A word x is an outfix of a word y if there is a word w such that $x_1wx_2 = y$ and $x = x_1x_2$. A set X of words is outfix-free if no word in X is an outfix of any other word in X. A polynomial-time algorithm was developed to determine outfix-freeness of regular languages. Furthermore, a linear-time algorithm that computed a prime outfix-free decomposition for outfix-free regular languages was given. There are also papers on theoretical issues related to the problem of formal language decomposition [11–16].

Let us note that all the works discussed above differ from our approach to solving the decomposition problem. Firstly, we consider finite languages that do not have to satisfy any specific conditions. Secondly, the parallel algorithm we propose is intended to compute all decompositions of a finite language L in the form of $L = L_1L_2$.

This paper builds on our previous efforts in developing the sequential and parallel algorithms for finite language decompositions. Let us briefly review the results obtained in those efforts. A sequential algorithm for finding the decomposition of a finite language was proposed in [17]. The algorithm returned only the first decomposition found of a given language. A threshold parameter T (see p. 12) that impacted the operation of the algorithm was kept constant, meaning it was not adaptively adjusted while the algorithm was running. The article introduced the concept of significant state along with the proof that for every composite language there is at least one decomposition based solely on significant states. In the experimental part of the paper, 240 languages of size less than 2000 words were studied, including 120 prime languages. The implementation was done in Python, and the average running time of the algorithm for test languages was in the order of a few seconds.

The approach for finding the first decoposition of a finite language by using selected meta-heuristics was discussed in [18]. The paper presented the results for the simulated annealing algorithm, tabu search, and genetic and randomized algorithms, all sequentially implemented in Python. Computational experiments were carried out on 1200 languages of sizes less than 2000 words with algorithm execution time limits of 10 and 60 seconds. Within these limits, the algorithms returned quite a lot of wrong answers for composite languages, claiming they were prime.

A basic parallel algorithm (its short description is included in Section 4) for finding all decompositions of a finite language using the concept of significant state was given in [19]. The algorithm consisted of two phases. In the first phase, each process executed the same code, and in the second phase the computation was spread across the processes available based on their ranks. The algorithm was implemented in the C language and Message Passing Interface (MPI). The experiments, conducted with up to 22 processes covered four languages with a word count between 800–6583 words.

A preliminary version of an adaptive parallel algorithm for solving the decomposition problem was given in [20]. The adaptive algorithm was based on the modified concept of significant state compared with that given in [17]. The algorithm consisted of two phases and included a method of pruning of the search space, and a simplified verification of prospective decomposition sets. It also involved an adaptive way for adjusting the threshold parameter T to keep the balance between the times spent in two phases of the algorithm. The experiments concerned nine languages up to 90000 words in size, solved with 32 processes in the run time of a few minutes. The algorithm was implemented in the C language and MPI interface.

To summarize, our previous work encompassed several sequential and parallel algorithms to solve the decomposition problem for finite languages. The sequential algorithms were able to decompose the languages of size up to 2000 words, while the parallel algorithms could tackle the languages of size up to 90000 words with 32 processes.

In the current paper we provide the advanced adaptive parallel algorithm, which can solve languages of size between 160000 and more than 200000 words in the run time of tens of minutes by using 128 processes. The results of comprehensive computational experiments on the variety of 4000 benchmark languages are also given. To the best of our knowledge, the parallel algorithm and in-depth experimental study of the problem under consideration have not been reported in the literature thus far.

3 Preliminaries

Below, we recall selected concepts from the theory of formal languages and automata. For more details the reader may refer to the textbooks [21, 22].

An *alphabet* Σ is a nonempty finite set of symbols. A word w is a sequence of zero or more symbols taken from Σ. The length of a word is denoted by $|w|$, with the special case of zero-length word being the *empty word* λ, for which $|\lambda| = 0$. A *prefix* of a word is any number of leading symbols of that word, and a *suffix* is any number of trailing symbols. By convention, Σ^* denotes the set of all words over an alphabet Σ, and Σ^+ denotes the set $\Sigma^* - \{\lambda\}$.

A *finite language* $L \subset \Sigma^*$ is a finite set of words $w \in \Sigma^*$. A finite language L is *trivial* if it consists of the empty word λ, that is $L = \{\lambda\}$, and it is *nontrivial* if it contains at least one nonempty word. Let $u, v, w \in \Sigma^*$. A *concatenation* of words u and v produces a word $w = uv$, where w is created by making a copy of word u and following it by a copy of word v. Let $U, V, W \subseteq \Sigma^*$. Then the concatenation (or product) of sets U and V produces a set $W = UV$ such that $W = \{w : w = uv, u \in U, v \in V\}$.

A *deterministic finite automaton* (DFA) is defined by a quintuple $A = (Q, \Sigma, \delta, s, Q_F)$, where Q denotes a finite set of automaton states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $s \in Q$ is the initial (start) state, and $Q_F \subseteq Q$ is a set of final (accepting) states. An automaton is *deterministic* iff for all states $q \in Q$ and symbols $a \in \Sigma$, $|\delta(q, a)| \leq 1$; in other words, each state q has at most one out-transition marked by a. A deterministic automaton is *finite* iff its state set Q is finite. Let us extend the transition function δ to words over Σ. Formally we define $\delta(q, \lambda) = q$, and for all words w and input symbols a, $\delta(q, wa) = \delta(\delta(q, w), a)$. So $\delta(q, w) = q'$ means that the word w takes A from the state q to the state q'.

A directed graph $G = (V, E)$, called a *transition diagram*, is associated with a finite automaton A, where $V = Q$, $E = \{p \overset{a}{\to} q \mid q = \delta(p, a)\}$, and $p \overset{a}{\to} q$ denotes an arc labeled a in the transition diagram. Given a word $w = a_1 a_2 \ldots a_m$, $a_i \in \Sigma$, $i \in [1, m]$, a *w path* is a sequence of transitions labeled with symbols of w in the transition diagram. A w path is denoted by a sequence of states $q_1, q_2, \ldots, q_{m+1}$ where $q_j \in Q$, $j \in [1, m+1]$, lying on the path. A deterministic automaton A accepts a word w if there is the w path $q_1 q_2 \ldots q_{m+1}$ leading from the initial state to a final state of A, so $q_1 = s$ and $q_{m+1} \in F$.

Define the *left* (resp. *right*) language of a state $q \in Q$ as $\overleftarrow{q} = \{w : \delta(s, w) = q\}$ (resp. $\overrightarrow{q} = \{w : \delta(q, w) \in Q_F\}$). Put simply, the left (resp. right) language of q consists of words w for which there is a w path from the initial state s to q (resp. from q to a final state).

A finite automaton $A = (Q, \Sigma, \delta, s, Q_F)$ is said to accept a language L when for each word $w \in L$ there is a w path beginning in the state s, and ending in a state $q \in Q_F$, or more formally $L = \{w|\delta(s, w) \in F\}$. Deterministic and nondeterministic[1] finite automata accept the same set of languages, namely the set of *regular languages*. Minimum-state acyclic DFAs accept the set of *finite languages*. Given all states $p, q \in Q$, $p \neq q$, a DFA is *minimum-state* iff $\overrightarrow{p} \neq \overrightarrow{q}$,[2] and it is *acyclic* iff $\delta(q, w) \neq q$ for every word $w \in \Sigma^+$ and state $q \in Q$.

In what follows, we only consider minimum-state acyclic deterministic automata, so each such automaton will be referred to as *automaton* herein.

Let L be a nontrivial finite language. A *decomposition* of L of index k where $k \geq 2$ is the family of languages L_i for $i \in [1, k]$ called *factors* that once concatenated give language L, so we have $L = L_1 L_2 \ldots L_k$. The decomposition is *nontrivial* if languages L_i are nontrivial, that is $L_i \neq \{\lambda\}$ for $i \in [1, k]$. Otherwise, the decomposition is *trivial*. In every nontrivial decomposition of a finite language L, the number of factors, k, is at most equal to the length of the longest word in L. Clearly, any language L has the trivial decompositions $L\{\lambda\}$ and $\{\lambda\}L$. In what follows, by a decomposition we always mean a nontrivial decomposition. A language L is called *prime* if L has no decomposition of index 2, otherwise it is called *composite* (or *decomposable*). The *prime decomposition* of L is a decomposition $L = L_1 L_2 \ldots L_k$, $k \geq 2$, where each language L_i for $i \in [1, k]$ is prime. It has been proven that every finite language is prime or has a prime decomposition, which is generally not true for infinite languages [13–16].

In this paper, we investigate the problem of finding all decompositions of index 2 of a nontrivial finite language L. For a particular decomposition we are looking for two nonempty finite languages L_1 and L_2 that once concatenated give language L, so we have $L = L_1 L_2$. Note that a given language L can have many decompositions of index 2.

Let us introduce the notion of a decomposition set that is suitable for the study of decompositions of regular languages.[3] The notion is related to the left quotients of regular languages. Let L be a regular language over an alphabet Σ, and let $A = (Q, \Sigma, \delta, s, Q_F)$ be the minimum-

[1] A state $q \in Q$ of a nondeterministic automaton may have more than one out-transition marked by a given symbol $a \in \Sigma$; then $|\delta(q, a)| > 1$.

[2] When two distinct states have the same right language, they could be merged into one, making a smaller deterministic automaton. Two states p, q can be merged giving a single state $(\overleftarrow{p} \cup \overleftarrow{q}, \overrightarrow{p})$ by combining their in-transitions and using the out-transition from just one of them.

[3] Recall that finite languages we deal with are regular.

state finite deterministic automaton accepting L. For any nonempty set $D \subseteq Q$, we define the left and right languages of D:[4]

$$L_1^D = \cup_{g \in D} \overleftarrow{g} \quad \text{and} \quad L_2^D = \cap_{g \in D} \overrightarrow{g}. \tag{1}$$

Note that languages L_1^D and L_2^D are regular as they are subsets of the regular language L.

Theorem 1 *Having L and A defined as above, assume that L is composite, so $L = L_1 L_2$ for regular languages L_1 and L_2. Define a set $D \subseteq Q$, called a decomposition set, by*

$$D = \{z \in Q \mid \delta(s, w) = z, \text{ for some } w \in L_1\}.$$

Then $L_1 \subseteq L_1^D$, $L_2 \subseteq L_2^D$ and

$$L = L_1^D L_2^D \tag{2}$$

is the decomposition of L into two regular languages.

Proof See [1, 16]. □

The decomposition $L = L_1^D L_2^D$ is referred to as decomposition *induced* by the set D. Theorem 1 implies that every decomposition of the regular language L is included in the decomposition of L induced by the decomposition set. The decomposition $L = L_1 L_2$ is said to be *included* in the decomposition $L = L_1' L_2'$ if $L_i \subseteq L_i'$ for $i = 1, 2$.

Corollary 1 *To solve the problem of finding all decompositions of index 2 of a nontrivial finite language L, we need to check (2) for all subsets D of Q. If none of these subsets induces a decomposition, we conclude that L is prime.*

It follows from the corollary that solving this problem is equivalent to solving the primality problem.

Problem 1 (*Primality*) Let L be a finite language over a finite alphabet Σ given as a DFA. Answer the question whether L is prime.

It was shown that the primality problem for finite languages is NP-hard [1], and for regular languages it is PSPACE-complete [23, 24].

Example 1 As an illustration of Theorem 1, consider a finite composite language $L = \{\lambda, a, b, aa, ab, aaa, aab, aaaa\}$ that has a single decomposition of index 2. The transition diagram for the minimum-state deterministic automaton A accepting language L is depicted in Fig. 1. Note that $L = L_1 L_2$, where $L_1 = \{\lambda, a\}$ and $L_2 = \{\lambda, b, aa, ab, aaa\}$.

Then, according to Theorem 1 there is the decomposition set $D \subseteq Q$ where $Q = \{s, q, r, p, t\}$ such that $L = L_1^D L_2^D$, $L_1 \subseteq L_1^D$ and $L_2 \subseteq L_2^D$. Indeed, in our case $D = \{s, q\}$, the left language L_1^D includes words w satisfying $\delta(s, w) \in D$: $L_1^D = \overleftarrow{s} \cup \overleftarrow{q} = \{\lambda\} \cup \{a\} = \{\lambda, a\}$, and the right language L_2^D includes words w satisfying $\delta(g, w) \in Q_F$ where $g \in D$: $L_2^D = \overrightarrow{s} \cap \overrightarrow{q} = \{\lambda, a, b, aa, ab, aaa, aab, aaaa\} \cap \{\lambda, a, b, aa, ab, aaa\} = \{\lambda, a, b, aa, ab, aaa\}$. Moreover, $L_1 = L_1^D$ and $L_2 \subset L_2^D$. Observe that words in L_1^D and L_2^D are prefixes and suffixes of L, and each word $w \in L$ is divided by at least one state $g \in D$ such that $w = xgy$ where x and y are a prefix and suffix of w, respectively. For example, considering words of L we have ($s, q \in D$): $\lambda s \lambda$, $\lambda s a$, $a q \lambda$, $\lambda s b$, $\lambda s a a$, $a q a$, $\lambda s a b$, $a q b$, $\lambda s a a a$, $a q a a$, $a q a b$, $a q a a a$.

Example 2 A finite language may have more than one decomposition of index 2. The following language $L = \{ab, aba, abb, bb, bba, bbb\}$ (Fig. 2) has two decompositions, the first defined by $D = \{p\}$, $L_1 = L_1^D = \overleftarrow{p} = \{a, b\}$, $L_2 = L_2^D = \overrightarrow{p} = \{b, ba, bb\}$, and the second by $D' = \{q\}$, $L_1{}' = L_1{}^{D'} = \overleftarrow{q} = \{ab, bb\}$, $L_2{}' = L_2{}^{D'} = \overrightarrow{q} = \{\lambda, a, b\}$.

Below we give a few examples of application of finite language decomposition.

Example 3 As already mentioned in Section 2, K. V. Hung considered factorization of supercodes [7]. He designed the algorithm to decompose a supercode L into prime components. The algorithm is based on the so-called bridge states, which are found in a non-returning and non-exiting acyclic deterministic finite automaton (N-ADFA) $\mathscr{A}$ accepting the code-words $w \in L$. An automaton $\mathscr{A}$ is non-returning if its start state has no in-transitions, and it is non-exiting if its final states have no out-transitions. A state b in automaton $\mathscr{A}$ is called a bridge state if b is neither the start state nor a final state, and each w path in $\mathscr{A}$ passes through b. Assume that $\mathscr{A}$ is the minimal N-ADFA accepting a supercode L that has k, $k \geq 1$, bridge states. Then L can be decomposed into $k + 1$ prime supercodes $L_1, L_2, \ldots, L_{k+1}$ such that $L = L_1 L_1 \ldots L_{k+1}$. The bridge states for a given automaton $\mathscr{A}$ can be identified in $O(|Q| + |\delta|)$ time where $|Q|$ and $|\delta|$ are the number of states and transitions of $\mathscr{A}$, respectively. As an example the supercode $L = \{ab^2ac, ab^5c, ac^3bac, ac^3b^4c\}$ is considered.[5] The

[4]The languages can be also written as $L_1^D = \{w \mid \delta(s, w) \in D\}$, and $L_2^D = \bigcap_{g \in D} \{w \mid \delta(g, w) \in Q_F\}$.

[5]A word a^i is a sequence of a's of length i.

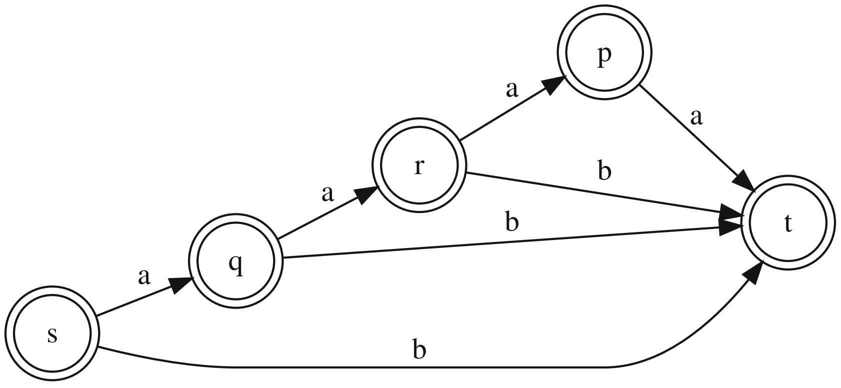

Fig. 1 Transition diagram for automaton accepting language L from Example 1

minimal N-ADFA $\mathscr{A}$ accepting L has four bridge states, so L may be decomposed uniquely into five prime supercodes: $L = \{a\}\{b, c^3\}\{b\}\{a, b^3\}\{c\}$ [7].

Instead of employing the notion of bridge states, one can readily find this factorization by calling recursively our parallel decomposition algorithm (described in Section 5). The sequence of calls gives the following: $L = L_1L_2$ where $L_1 = \{ab, ac^3\}$, $L_2 = \{bac, b^4c\}$; then $L_1 = L_1^1L_2^1$ where $L_1^1 = \{a\}$, $L_2^1 = \{b, c^3\}$; $L_2 = L_1^2L_2^2$ where $L_1^2 = \{b\}$, $L_2^2 = \{ac, b^3c\}$; and finally $L_2^2 = L_1^3L_2^3$ where $L_1^3 = \{a, b^3\}$, $L_2^3 = \{c\}$. In general, to find the prime decomposition of a supercode consisting of n prime components, at most n calls of the decomposition algorithm are needed.

Example 4 Deterministic finite automata (DFAs) are of importance in many fields of science. They also have many practical applications—in the design of compilers and text processors, in natural language processing, speech recognition, among others. One of the concerns regarding DFAs is the memory efficient storage of their representation. A DFA $\mathscr{A}$ can be represented by the list of words of a language L (assuming it is finite) accepted by $\mathscr{A}$. Such representation can be memory intensive if L is large, therefore a better option is to decompose L and store its factors L_1 and L_2. Table 1 shows the reduction of size of languages (expressed in word count) from Section 6, obtained by using the decomposition based option. As can be seen, the reduction rate exceeds 99% for all languages in the table.

Example 5 A company signed the contract to supply the pipes with lengths 30, 31, 32, 36, 37, and 38 m. For technical reasons, the company can manufacture pipes of any length, but no greater than 30 m. A longer pipe can be produced by welding shorter pipes. The main part of the pipe production cost is to prepare a mold of the given length, so the number of molds needed to complete the contract should be minimized. The question is, which lengths should have the molds that must be prepared to implement the supply contract mentioned above. To answer this question, let us define the set of words $L = \{a^{30}, a^{31}, a^{32}, a^{36}, a^{37}, a^{38}\}$.

Then, among all decompositions $L = L_1L_2$, we look for the one for which the size of set $L_1 \cup L_2$ is minimal, and each word in L_1 and L_2 is no longer than 30. The solution is the decomposition $L = \{a^{15}, a^{16}, a^{17}\}\{a^{15}, a^{21}\}$, so we need only four molds of lengths 15, 16, 17, and 21 m.

Example 6 Given the sets of words S_+ and S_-, called examples and counterexamples, find a context-free grammar $G = (V, \Sigma, P, S)$ such that $S_+ \subseteq \mathscr{L}(G)$, and $S_- \cap \mathscr{L}(G) = \emptyset$ where $\mathscr{L}(G)$ denotes the language generated by G. The basic decomposition algorithm (described in Section 4) can be readily modified to find an incomplete decomposition that allows a finite language L to be written as $L = L_1L_2 \cup R$ where concatenation L_1L_2 is the greatest possible, and R is the set of other words belonging to L.

Let $S_+ = \{ab, ba, aabb, abab, baba, bbaa, baab, abba\}$ be the set of examples, and let $S_- = \{a, b\}^d - S_+$ with $d \leq 4$ be the set of counterexamples. Based on these sets the context-free grammar can be found in two stages. In the first stage, using the modified basic decomposition algorithm, we create a series of incomplete decompositions. In particular, we get $S_+ = LL \cup R$ where $L = \{ab, ba\}$, and $R = \{ab, ba, aabb, bbaa\}$; then $R = AB \cup F$ where $A = \{a\}$, $B = \{b, abb\}$, and $F = \{ba, bbaa\}$; and further $F = HI$ where $H = \{b\}$, and $I = \{a, baa\}$, and so on. This procedure is iterated for the subsequent sets of words until a grammar in the Chomsky normal form is obtained (a

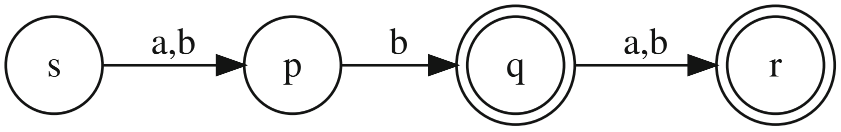

Fig. 2 Transition diagram of automaton A accepting language L from Example 2

Table 1 Storage reduction R obtained by replacing language L by its factors, $L = L_1L_2$

$\|L\|$	$\|L_1\|$	$\|L_2\|$	$\|L_1\| + \|L_2\|$	R [%]
60486	149	418	567	99.1
69858	292	244	536	99.2
150864	489	313	802	99.5
159239	331	495	826	99.5
169477	507	339	846	99.5
172221	393	447	840	99.5
194390	592	332	924	99.5
214994	447	486	933	99.6
217183	482	462	944	99.6
220482	468	475	943	99.6
230512	665	348	1013	99.6

context-free grammar G not generating the empty word λ is said to be in Chomsky normal form if all of its production rules are of the form $A \to BC$ or $A \to a$ where A, B, and C are nonterminal symbols and a is a terminal symbol). The sets S_+, L, R, A, B, F, H, I, etc., are represented by grammar variables, and each set concatenation (decomposition) corresponds to the grammar production rule. The second stage simplifies the grammar through examining whether each pair of grammar variables can be merged. It can be done if after such merger the grammar does not accept any word from the set S_-. Furthermore, the unit production rules are eliminated. For the sets S_+ and S_- specified above, we get the following context-free grammar:

$$\begin{array}{ll} S \to S\,S \mid A\,B \mid B\,A & A \to a \\ B \to b \mid A\,C \mid B\,S & C \to B\,B \end{array}$$

which defines the infinite language of words having the same number of symbols a and b. For more details, see [25, p. 71], [26, 27].

4 Basic parallel decomposition algorithm

In this section we outline the basic parallel algorithm (or shortly, basic algorithm) for finite language decomposition [19]. It has been a starting point for devising the adaptive parallel decomposition algorithm (or, adaptive algorithm). The pseudocode of the basic algorithm expressed as a recursive procedure DECOMPOSE(W, D) is shown in Fig. 3.

Let $A = (Q, \Sigma, \delta, s, Q_F)$ be the minimum-state acyclic DFA[6] accepting an input language $L = \{w_i\}$, $i \in [1, n]$. The basic algorithm explores the set of states Q to find the decomposition sets D where $D \subseteq Q$. According to Theorem 1 each decomposition set D induces a decomposition $L = L_1^D L_2^D$. The algorithm finds all decompositions of the input language L by employing an exhaustive search of Q with pruning.

Let W be a set of pairs (w_i, Q_{w_i}) where the word $w_i = a_1a_2 \ldots a_m$, $w_i \in L$, $a_j \in \Sigma$, $j \in [1, m]$, and let $Q_{w_i} = \{q_1, q_2, \ldots, q_{m+1}\}$, be a set of states $q_k \in Q$, $k \in [1, m+1]$ lying on the w_i path where $q_1 = s$, and $q_{m+1} \in Q_F$. Let STATES(W) be the function that returns a set of states appearing in all sets Q_{w_i} for $w_i \in L$, and let MINSTATES(W) be the function that returns a pair $(w_i, Q_{w_i}) \in W$ in which set Q_{w_i} of states dividing w_i, is minimal.

Assume that D is a decomposition set to be found (see *Phase1* in Fig. 3). Initially this set is empty, and it is gradually built up as the basic algorithm processes the words $w_i \in L$. The words to be processed are selected in order of increasing sizes of their sets Q_{w_i} by using the function MINSTATES(W). Each state $q \in Q$ that divides a word $w_i \in L$ into two parts is inserted in set D, which is then checked to see if it is a decomposition set. The basic algorithm is recursive, so each subset of states in Q that can be a candidate for a decomposition set is examined.

Definition 1 Suppose a word $w \in L$ is divided by a state $q \in Q$ into two parts, so $w = xy$ where $x \in \overleftarrow{q}$ and $y \in \overrightarrow{q}$, and suppose the state q is inserted in set D. Then each state $r \in$ STATES(W) for which $y \notin \overrightarrow{r}$ is considered redundant (or significant otherwise).

In words, suppose $w = xy$. If state $q \in Q$ is inserted in D then each state $r \in Q$ that does not have y in its suffix set becomes redundant. Note that a given state r may be either redundant or significant depending on the states that are in set D in the current recursive execution of the algorithm.

Lemma 1 *Let D where $D \subseteq Q$ be the decomposition set for a finite language L. Then each state $q \in D$ is significant.*

Proof If D is a decomposition set for L, then each $w \in L$ is divided into two parts by at least one state $q \in D$, that is $w = xy$, $x \in \overleftarrow{q}$ and $y \in \overrightarrow{q}$. Once state q is inserted in D, suffix y appears in the right language L_2^D. The definition $L_2^D = \cap_{r \in D} \overrightarrow{r}$ involves the intersection of sets $\overrightarrow{q}$, thus suffix y must be in a set $\overrightarrow{r}$ for each $r \in D$. If a state r does not satisfy this condition, it is redundant and can be omitted during further recursive search. To conclude, only significant states can occur in a decomposition set [17]. □

Let SUFFIX(w, q) be the function that returns suffix y of a word $w = xy$ that is divided by a state q. Suppose the set D has already been built based on words w_1, w_2, $\ldots$, w_{i-1}, and we want to extend D for a word w_i with $Q_{w_i} = \{q_1, q_2, q_3, \ldots, q_{m+1}\}$ for some m. With this aim,

[6] There are several algorithms of time complexity $O(\sum_{w_i \in L} |w_i|)$ to construct such automaton for a given language L; see for example [28].

Fig. 3 Basic algorithm

```
▷ Construct minimum-state automaton $A = (Q, \Sigma, \delta, s, Q_F)$ accepting language $L$
▷ and using $A$ create structure $W = \{(w_i, Q_{w_i})\}$ for $i = [1, n]$
procedure DECOMPOSE(in $W$, in $D$)                              ▷ initially $D = \emptyset$
    if $|\text{STATES}(W) - D| > T$ then                         ▷ carry out recursive search for sets $D$
        ▷ begin Phase1
        $(w, Q_w) \leftarrow \text{MINSTATES}(W)$               ▷ find word $w$ with minimum $|Q_w|$
        if $Q_w = \emptyset$ then                                ▷ word $w$ is nondivisible
            return
        else
            $W \leftarrow W - (w, Q_w)$
            for all $q \in Q_w$ do                               ▷ consider all states $q$ dividing word $w$
                $\text{REMOVERED}(W, \text{SUFFIX}(w, q))$       ▷ prune state space
                $\text{DECOMPOSE}(W, D \cup \{q\})$              ▷ extend $D$ and continue search
            end for
        end if
        ▷ end Phase1
    else                        ▷ threshold for size of set $\mathscr{Q}$ reached: $|\text{STATES}(W) - D| \leqslant T$
        ▷ begin Phase2
        $D_1 \leftarrow D$                                       ▷ save partial decomposition set $D$ obtained in Phase1
        for $j := 0$ to $2^{|\mathscr{Q}|} - 1$ do              ▷ spread sets $D \cup C_j$ across $\pi$ processes
            if $j \bmod \pi = r$ then                            ▷ subset $C_j$ is handled by process $\mathscr{P}_r$
                Create subset $C_j \subseteq \mathscr{Q}$        ▷ $\mathscr{Q} = \text{STATES}(W) - D$
                $D \leftarrow D \cup C_j$                        ▷ complement set $D$ with subset $C_j$
                if $L = L_1^D L_2^D$ then                        ▷ set $D$ is verified by process $\mathscr{P}_r$
                    $\text{OUTPUT}(D)$                           ▷ decomposition set $D$ found
                end if
                $D \leftarrow D_1$                               ▷ restore partial decomposition set $D$
            end if
        end for
        ▷ end Phase2
    end if
end procedure
```

we first select state q_1 that divides w_i ($w_i = x_i^1 y_i^1$), add q_1 to D, and remove redundant states (based on Definition 1) from all sets $Q_{w_k} \in W$ for $k = i+1, i+2, \ldots, n$, using the procedure REMOVERED(W, y) with $y = y_i^1$ (Fig. 4). Once the recursive call DECOMPOSE($W \cup \{q_1\}$) completes, we carry out the above operations for states $q_2, q_3, \ldots, q_{m+1}$, and different divisions $w_i = x_i^2 y_i^2$, $w_i = x_i^3 y_i^3$, etc. Due to the removal of redundant states, the number of states in structure W decreases in subsequent recursive executions of procedure DECOMPOSE(W, D). To sum up, *Phase1* of the basic algorithm builds up the decomposition set D by processing words $w_1, w_2, \ldots$, and so on, while downscaling the set $\mathscr{Q} = \text{STATES}(W) - D$ of candidates to extend the set D.

The basic algorithm shown in Fig. 3 is run by a set of sequential processes $\mathscr{P}_r$, $r \in [0, \pi - 1]$, where r is the rank (index) of a process, and π is the number of processes available. Each process $\mathscr{P}_r$ executes the code of procedure DECOMPOSE(W, D), and all processes in the set are running in parallel.

A process is executed by a conventional processor, or core of a multi-core processor (from now on, we will use term *processor* for both of these computing devices).

The execution of the basic algorithm in each process consists of two phases. In *Phase1* subsequent sets D are established, which are then processed in *Phase2*. The code executed in *Phase1* is the same in all processes, while in *Phase2* the computations performed by processes differ from each other.

As mentioned above, *Phase1* builds gradually decomposition sets D while reducing sets $\mathscr{Q}$ containing states, which are candidates to extend sets D. Notice that *Phase1* does not determine the complete set D but only a *partial* set $\mathscr{D}$ where $\mathscr{D} \subset D$.[7] When $\mathscr{Q}$ becomes small enough due to the removal of redundant states, that is $|\mathscr{Q}| \leq T$ for some threshold T, the algorithm moves to *Phase2* in which each process takes a collection of *prospective* sets $\mathscr{D} \cup C_j$ to verify as to whether they constitute decomposition sets for L, where each C_j is a subset of $\mathscr{Q}$ for $j \in [0, 2^{|\mathscr{Q}|} - 1]$.

More specifically, process $\mathscr{P}_0$ takes sets $\mathscr{D} \cup C_j$ with $C_j = C_0, C_\pi, C_{2\pi}, \ldots$, process $\mathscr{P}_1$ sets $\mathscr{D} \cup C_j$ with $C_j = C_1, C_{\pi+1}, C_{2\pi+1}, \ldots$, etc. The advantage of such an arrangement is that each collection of sets $\mathscr{D} \cup C_j$ can be verified separately from other collections. As a result, it allows the work related to verifying sets $\mathscr{D} \cup C_j$ to be readily spread across processes $\mathscr{P}_r$.

In order to boost performance by making the most of parallel computing capabilities, the basic algorithm controls the depth of recursion in *Phase1*. For this purpose, a threshold T, already mentioned, is imposed on the size of set $\mathscr{Q}$ containing candidate states to extend set $\mathscr{D}$ built to date. The threshold T is a parameter of the algorithm (the parameter is adjusted at runtime in the adaptive algorithm, see Section 5). When the size of $\mathscr{Q}$ becomes reasonably small as a result of the downscaling process, then instead of

[7]The partial set $\mathscr{D}$ is denoted by D in the code of the basic algorithm in Fig. 3.

Fig. 4 Procedure RemoveRed

```
1: procedure RemoveRed(in out $W$, in $y$)        ▷ remove redundant states from $W$
2:     for all $r \in$ States($W$) do              ▷ $y$ is suffix of word $w$
3:         if $y \notin \overrightarrow{r}$ then
4:             Remove state $r$ from each set $Q_w$ of $W$     ▷ state $r$ is redundant
5:         end if
6:     end for
7: end procedure
```

looking for the complete decomposition set on a deep level of recursion in *Phase1*, the processes move to *Phase2* to verify sets $\mathscr{D} \cup C_j$. It is worth noting that while running the basic algorithm the processes do not communicate with one another. They only synchronize their action at the beginning of computation to enter the input language, and at the end of computation when the results obtained by the processes are collected.

Let us consider the average time complexity of basic algorithm, $\mathscr{T}_b(\pi, n)$. Let $d(n, T)$ be the average number of sets $\mathscr{D}$ found in *Phase1*. Let $t_1(n, T)$ be the average time to find a single set $\mathscr{D}$ in *Phase1*, and let $t_2(n, \psi T)$ be the average time to verify as to whether a set $\mathscr{D} \cup C_j$, $j = 0, 1, \ldots, 2^{\psi T} - 1$, is a final decomposition set for L in *Phase2*. The value of ψT determines the average size of sets $\mathscr{Q}$ processed in *Phase2* where $\psi \in (0.0, 1.0]$. Considering the above, the average time complexity of basic algorithm is as follows:[8]

$$\mathscr{T}_b(\pi, n) = d(n, T) \cdot (t_1(n, T) + (t_2(n, \psi T) \cdot 2^{\psi T})/\pi). \quad (3)$$

There are two components in this equation: the first, $d(n, T) \cdot t_1(n, T)$, determines the total average run time of *Phase1*, and the second, $d(n, T) \cdot (t_2(n, \psi T) \cdot 2^{\psi T})/\pi$, the total average run time of *Phase2*. For a fixed value of n, the run time of *Phase2* grows exponentially as a function of ψT. This growth is due to the exponential number of prospective decomposition sets $\mathscr{D} \cup C_j$ to be verified in *Phase2*.

5 Adaptive parallel decomposition algorithm

With the aim of improving the basic algorithm, we propose several refinements. The first three refinements introduce into the adaptive algorithm the effective methods for pruning the search space (Fig. 5). The fourth refinement adjusts threshold T while the adaptive algorithm is executed, based on the runtime acquired data related to the performance of *Phase1*.

Let us discuss the refinements in more detail. The first refinement concentrates on removing the redundant states $q \in Q$ of the automaton accepting the input language L. The redundant states have already been eliminated in the basic algorithm (see procedure RemoveRed(W, y)). We extend the scope of such an elimination in the adaptive algorithm.

Definition 2 A state $q \in Q_w$, $Q_w \subseteq Q$, is considered *redundant* (or *significant* otherwise) if for a word $w = xy$, $w \in L$, $x \in \overleftarrow{q}$, and $y \in \overrightarrow{q}$, the following holds:

$$U(y) \cdot |\overrightarrow{q}| < |L| \quad (4)$$

where $U(y)$ denotes the number of occurrences of suffix y in all words $w \in L$.

To justify (4), suppose q is the only state in the decomposition set D defined in Theorem 1. Then, given word $w = xy$ that is divided by state q, the values of $U(y)$ and $|\overrightarrow{q}|$ determine the sizes of sets $L_1^D = \cup_{q \in D} \overleftarrow{q}$ and $L_2^D = \cap_{q \in D} \overrightarrow{q}$, respectively. In fact, the value of $U(y)$ is the number of prefixes x belonging to set L_1^D, since $U(y)$ is counted over all words $w = xy$, $w \in L$, with prefix x followed by suffix y. In view of the above, the product $U(y) \cdot |\overrightarrow{q}|$ on the left side of (4) determines the upper limit of the number of words that could be created by concatenating sets L_1^D and L_2^D. Now, if this product is less than $|L|$, then $L = L_1^D L_2^D$ is not satisfied, which means that D cannot be the decomposition set for L. So the state $q \in D$ is redundant.

Elimination of redundant states $q \in Q_w$ satisfying (4) is implemented in the procedure RemoveRed2(W, y) (Fig. 6). When one or more states are found redundant and removed from sets $Q_w \in W$, which is indicated by variable f, it can cause a decrease in the number of occurrences of suffix y, given by $U(y)$, and also in the size of right language $\overrightarrow{q}$. As a result, more redundant states can be removed from sets Q_w.

Using (4), we can also remove redundant states before the adaptive algorithm begins. Once automaton A has been constructed, the procedure BuildW(A, W) (Fig. 7) builds structure W, which is the input parameter to procedure Decompose2(W, D) (Fig. 5). While creating sets Q_w only significant states of Q are considered. Hence, the number of states in W that are then processed by the adaptive algorithm is smaller than the number of states in A.

Example 7 To clarify how structure $W = \{(w_i, Q_{w_i})\}$ is built, consider the language $L = \{a, aaa, aab, b\}$ (Fig. 8).

The suffixes of words $w_i \in L$ along with their frequency counts in L are $(\lambda, 4)$, $(a, 2)$, $(aaa, 1)$, $(aa, 1)$, $(aab, 1)$, $(ab, 1)$, $(b, 2)$, and the sizes of right languages are $|\overrightarrow{q0}| = 4$,

[8] Equation (3) defines the average time complexity of the algorithm solving composite languages. In case of prime languages, a partial decomposition set may not exist, and then $d(n, T) = 0$.

Fig. 5 Adaptive algorithm

```
▷ Construct minimum-state automaton $A = (Q, \Sigma, \delta, s, Q_F)$ accepting language $L$
▷ Invoke procedure BUILDW of Fig. 7 to create structure $W$
procedure DECOMPOSE2(in $W$, in $D$)                        ▷ initially $D = \emptyset$
    ADJUSTT($|\text{STATES}(W) - D|$)                         ▷ adjust threshold $T$
    if $|\text{STATES}(W) - D| > T$ then                      ▷ carry out recursive search for sets $D$
        ▷ begin Phase1
        $(w, Q_w) \leftarrow \text{MINSTATES}(W)$             ▷ find word $w$ with minimum $|Q_w|$
        if $Q_w = \emptyset$ then                            ▷ word $w$ is nondivisible
            return
        else
            $W \leftarrow W - (w, Q_w)$
            for all $q \in Q_w$ do                          ▷ consider all states $q$ dividing word $w$
                REMOVERED2($W, \text{SUFFIX}(w, q)$)          ▷ prune state space
                DECOMPOSE2($W, D \cup \{q\}$)                 ▷ extend $D$ and continue search
            end for
        end if
        ▷ end Phase1
    else          ▷ threshold for size of set $\mathcal{Q}$ reached: $|\text{STATES}(W) - D| \leqslant T$
        ▷ begin Phase2
        $D_1 \leftarrow D$                                  ▷ save partial decomposition set $D$ obtained in Phase1
        $c \leftarrow \text{COMPLOWBOUND}(W, D)$              ▷ $c$ is lower bound for $C_j$ subsets
        for $j := 0$ to $2^{|\mathcal{Q}|} - 1$ do           ▷ spread sets $D \cup C_j$ across $\pi$ processes
            if $j \bmod \pi = r$ then                       ▷ subset $C_j$ is handled by process $\mathcal{P}_r$
                Create subset $C_j \subseteq \mathcal{Q}$          ▷ $\mathcal{Q} = \text{STATES}(W) - D$
                if $|C_j| \geqslant c$ then                   ▷ size of $C_j$ consistent with lower bound $c$
                    $D \leftarrow D \cup C_j$                  ▷ complement set $D$ with subset $C_j$
                    if VERIFY($D$) then                     ▷ set $D$ is verified by process $\mathcal{P}_r$
                        OUTPUT($D$)                         ▷ decomposition set $D$ found
                    end if
                    $D \leftarrow D_1$                         ▷ restore partial decomposition set $D$
                end if
            end if
        end for
        ▷ end Phase2
    end if
end procedure
```

Fig. 6 Procedure REMOVERED2 – improved version of REMOVERED

```
procedure REMOVERED2(in out $W$, in $y$)                    ▷ remove redundant states from $W$
    $f \leftarrow$ false                        ▷ $f$ shows whether any state is removed from sets $Q_w$
    for all $r \in \text{STATES}(W)$ do
        if $y \notin \overrightarrow{r}$ then
            Remove state $r$ from each set $Q_w$ of $W$, $f \leftarrow$ true
        end if
    end for
    if $f =$ true then
        for all $(w, Q_w) \in W$ and all $r \in Q_w$ do          ▷ for all $q$ dividing word $w$
            $y \leftarrow \text{SUFFIX}(w, r)$
            if $U(y) \cdot |\overrightarrow{r}| < |L|$ then                ▷ testing Eq. (4)
                $Q_w \leftarrow Q_w - \{r\}$                 ▷ state $r$ is redundant, remove it from $Q_w$
            end if
        end for
    end if
end procedure
```

Fig. 7 Procedure BUILDW

```
procedure BUILDW(in $A$, in out $W$)                  ▷ build structure $W$ using automaton $A$
    for all $(w \in L)$ and $(i \in [1, \overline{w}])$ do          ▷ $\overline{w}$ denotes length of word $w$
        $S \leftarrow S \cup w[i, \overline{w}]$                     ▷ add suffix $w[i, \overline{w}]$ to set S
    end for
    for all $x \in S$ do                                   ▷ count suffix occurrences
        $U(x) \leftarrow$ number of occurrences of suffix $x$ in all words of $L$
    end for
    for all $w \in L$ do
        $Q_w \leftarrow \emptyset$, $W \leftarrow W \cup (w, Q_w)$, $q \leftarrow s$         ▷ $s \in Q$ is start state of $A$
        for $i := 1$ to $\overline{w}$ do
            if $U(w[i, \overline{w}]) \cdot |\overrightarrow{q}| \geq |L|$ then     ▷ $q$ is significant by Eq. (4)
                $Q_w \leftarrow Q_w \cup \{q\}$                    ▷ add $q$ to $Q_w$
            end if
            $q \leftarrow \delta(q, w[i])$                 ▷ move to next state in automaton $A$
        end for
    end for
end procedure
```

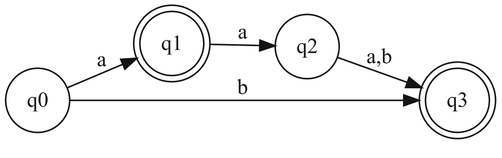

Fig. 8 Transition diagram of automaton accepting language $L = \{a, aaa, aab, b\}$

$|\overrightarrow{q1}| = 3$, $|\overrightarrow{q2}| = 2$ (we omit state $q3$, because it leads to the trivial decomposition $L = L\lambda$). To establish a pair (w_i, Q_{w_i}), we need to check (4) for word w_i. Let us check this equation for $w_i = aab$, which is divided by states $q0$, $q1$, and $q2$. The suffixes and checks are aab, ab, b, and $U(aab) \cdot |\overrightarrow{q0}| = 4 \geq |L|$, $U(ab) \cdot |\overrightarrow{q1}| = 3 < |L|$, and $U(b) \cdot |\overrightarrow{q2}| = 4 \geq |L|$. Thus, states $q0$ and $q2$ are significant, while state $q1$ is redundant. Continuing the similar analysis for the remaining words $w_i \in L$, we end up with $W = \{(a, \{q0, q1\}), (aaa, \{q0, q2\}), (aab, \{q0, q2\}), (b, \{q0\})\}$. Note, that the decomposition set for L is $D = \{q0, q2\}$ with $L_1^D = \{\lambda, aa\}$ and $L_2^D = \{a, b\}$.

The second refinement implements a method for reducing the search space by skipping the verification, if possible, of prospective decomposition sets $D = \mathscr{D} \cup C_j$. The verification is carried out in the basic algorithm by checking the condition $L = L_1^D L_2^D$ (Fig. 3). To make this verification more efficient, we determine the upper bound size of the language generated by set $\mathscr{D} \cup C_j$ (7). If the size of the input language L exceeds this bound, then we can omit the verification of $\mathscr{D} \cup C_j$.

Lemma 2 *Let $A = (Q, \Sigma, \delta, s, Q_F)$ be the automaton accepting a finite language L, let $D \subseteq Q$ be the final decomposition set for L, and let sets L_1^D and L_2^D be defined as in* (1). *Then the upper bound size of set $L_1^D L_2^D$ is:*

$$|L_1^D L_2^D| \leqslant |L_1^D| \cdot \min_{q \in D} |\overrightarrow{q}| \tag{5}$$

where $|L_1^D|$ is the size of L_1^D, and $\min_{q \in D} |\overrightarrow{q}|$ is the minimum size of the right language for states $q \in D$.

Proof The lemma follows directly from the definition of set $L_2^D = \bigcap_{q \in D} \overrightarrow{q}$. The size of L_2^D defined as the intersection of right languages $\overrightarrow{q}$ cannot be greater than $\min_{q \in D} |\overrightarrow{q}|$. □

Lemma 3 *The necessary condition for a finite language L to be decomposed by set $D \subseteq Q$ is:*

$$|L| \leqslant |L_1^D L_2^D|. \tag{6}$$

Proof Suppose we have $L = L_1 L_2$. By Theorem 1, $L_i \subseteq L_i^D$ for $i = 1, 2$. Hence, $L \subseteq L_1^D L_2^D$, and then $|L| \leqslant |L_1^D L_2^D| \leqslant |L_1^D| \cdot |L_2^D|$. □

Combining (5) and (6) we get the upper bound for $|L|$:

$$|L| \leqslant |L_1^D| \cdot \min_{q \in D} |\overrightarrow{q}| \tag{7}$$

which makes it possible to verify as to whether D can be the final decomposition set. The full verification first requires computing the sets L_1^D and L_2^D, then concatenating them, and finally checking whether $L = L_1^D L_2^D$. However, if (7) does not hold, then these operations can be avoided. The procedure VERIFY(D) (Fig. 9) performs a double-check of the constraints related to the upper bound size of set D. Both checks may result in the rejection of set D. We implement them consecutively because the cost of the first check is lower than the cost of the second one.

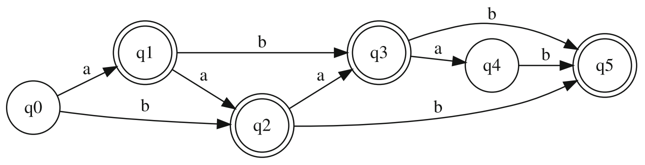

Fig. 10 Transition diagram of DFA accepting language L from Example 8

Fig. 9 Procedure VERIFY

```
procedure VERIFY(in $D$)
    $|L_1^D| \leftarrow \sum_{q \in D} |\overleftarrow{q}|$    ▷ estimate size $|L_1^D|$ of language generated by $q \in D$
    if $|L| \leqslant |L_1^D| \cdot \min_{q \in D} |\overrightarrow{q}|$ then    ▷ $|L|$ within bound defined in Eq. (7)
        Compute set $L_2^D$, and its size $|L_2^D|$, defined in Eq. (1)
        if $|L| \leqslant |L_1^D| \cdot |L_2^D|$ then    ▷ $|L|$ within bound defined in Eq. (6)
            Compute set $L_1^D$ defined in Eq. (1)
            if $L = L_1^D L_2^D$ then    ▷ perform final test on $D$
                return true    ▷ decomposition set $D$ found
            else
                return false    ▷ $L \neq L_1^D L_2^D$
            end if
        else    ▷ $|L|$ exceeds upper bound of $|L_1^D| \cdot |L_2^D|$
            return false
        end if
    else    ▷ $|L|$ exceeds upper bound of $|L_1^D| \cdot \min_{q \in D} |\overrightarrow{q}|$
        return false
    end if
end procedure
```

Fig. 11 Procedure COMPLOWBOUND

```
1: procedure COMPLOWBOUND(in $W$, in $D$)
2:     $|L_1^{C_j}| \leftarrow |L|/\min_{q\in D}|\overrightarrow{q}| - |L_1^D|$        ▷ compute lower bound of $L_1^{C_j}$ size
3:     Define $S$ : array[1..$|\mathscr{Q}|$] of integer        ▷ $\mathscr{Q} = \text{STATES}(W) - D$
4:     Put into array $S$, in descending order, sizes of left languages
5:         $|\overleftarrow{q}|$ generated by states $q \in \mathscr{Q}$
6:     $S \leftarrow \text{PREFIXSUM}(S)$        ▷ compute $S[i] = S[i-1] + S[i]$ for $i \in [2..|\mathscr{Q}|]$
7:     Find index $c$ in array $S$ such that $S[c] \geqslant |L_1^{C_j}|$
8:     return $c$        ▷ return lower bound size of $C_j$
9: end procedure
```

Example 8 To illustrate the procedure VERIFY let us examine language $L = \{a, aa, aaa, aaab, aaaab, aab, ab, abab, abb, b, ba, baab, bab, bb\}$ (Fig. 10). Assume that the adaptive algorithm moves from *Phase1* to *Phase2* with $\mathscr{D} = \{q0\}$ and $\mathscr{Q} = \{q1, q2\}$ (such assignments are made when the considered word a is divided by state $q0$). Suppose the procedure VERIFY is called with $D = \{q0, q2\}$. Then $|L_1^D| = 3$ ($L_1^D = \{\lambda, aa, b\}$), and $\min_{q\in D}|\overrightarrow{q}| = 5$ ($\overrightarrow{q0} = L$ and $\overrightarrow{q2} = \{\lambda, a, aab, ab, b\}$). So the result of the first check: $|L| \leqslant |L_1^D| \cdot \min_{q\in D}|\overrightarrow{q}|$ is positive as $14 \leq 3 \cdot 5$. The size $|L_2^D| = 4$ as $\overrightarrow{q0} \cap \overrightarrow{q2} = \{a, aab, ab, b\}$. Thus the condition $|L| \leqslant |L_1^D| \cdot |L_2^D|$ in the second check is not met because it holds $14 > 3 \cdot 4$. Therefore $D = \{q0, q2\}$ is not a decomposition set for L. Due to double-checking of conditions in the procedure under consideration, we do not need to concatenate sets L_1^D and L_2^D, nor compare the concatenation result with L. The decomposition set for the language in question is $D = \{q1, q2, q3\}$, and $L_1 = \{a, b, aa, ab, ba, aaa\}$, $L_2 = \{\lambda, b, ab\}$.

The approach taken in the third refinement is similar to that of the second refinement. We have established the lower bound size of subsets C_j (Lemma 4), which complete the partial decomposition set $\mathscr{D}$. A subset C_j—and consequently a set $\mathscr{D} \cup C_j$ as well—can be disregarded when the size of C_j is below the lower bound.

Lemma 4 *Let $\mathscr{D}$ be a partial decomposition set for L. Let $C_j \subseteq \mathscr{Q}$ be an arbitrary subset of candidate states to extend $\mathscr{D}$. Let $\mathscr{D} \cup C_j$ be a prospective decomposition set for L. Then*

$$|L| \leqslant |L_1^{\mathscr{D}} \cup L_1^{C_j}| \cdot \min_{q\in\mathscr{D}} |\overrightarrow{q}|. \tag{8}$$

Proof By Lemma 3, $|L| \leq |L_1^{\mathscr{D}\cup C_j} L_2^{\mathscr{D}\cup C_j}|$, and by Lemma 2, $|L_1^{\mathscr{D}\cup C_j} L_2^{\mathscr{D}\cup C_j}| \leq |L_1^{\mathscr{D}} \cup L_1^{C_j}| \cdot \min_{q\in\mathscr{D}}|\overrightarrow{q}|$, as $\mathscr{D} \cup C_j$ is a prospective decomposition set for L. □

From (8) we can derive the lower bound of $L_1^{C_j}$ size for an arbitrary subset C_j:

$$|L_1^{C_j}| \geqslant |L|/\min_{q\in\mathscr{D}} |\overrightarrow{q}| - |L_1^{\mathscr{D}}|. \tag{9}$$

Based on (9) the procedure COMPLOWBOUND (Fig. 11) helps to reduce the number of subsets $C_j \subseteq \mathscr{Q}$, $j \in [0, 2^{|\mathscr{Q}|} - 1]$, which are verified in *Phase2* of the adaptive algorithm. Recall that set $\mathscr{Q} = \text{STATES}(W) - \mathscr{D}$ includes candidate states to extend the partial decomposition set $\mathscr{D}$ obtained in *Phase1*. The set $L_1^{C_j}$ occurring on the left side of (9) is the sum of left languages: $L_1^{C_j} = \bigcup_{q\in C_j} \overleftarrow{q}$ where $C_j \subseteq \mathscr{Q}$ (see (1)). The procedure COMPLOWBOUND computes the minimum cardinality of C_j such that the sum of sizes of left languages generated by states $q \in C_j$, determined by the function PREFIXSUM, is greater than or equal to the value appearing on the right-hand side of (9). Using the required minimum cardinality of a subset C_j, it is either processed or discarded from further analysis in *Phase2*.

Example 9 To illustrate procedure COMPLOWBOUND consider language $L = \{a, aa, aaa, aaab, aaaab, aab, ab, abab, abb, b, ba, baab, bab, bb, bbab, bbb\}$ (Fig. 12). Suppose we enter *Phase2* with $\mathscr{D} = \{q3\}$ and $\mathscr{Q} = \{q1, q2, q6\}$ (such assignments result from dividing word aaa by state $q3$). Then we have $\min_{q3\in\mathscr{D}}|\overrightarrow{q3}| = 3$ (based on words λ, ab, b) and $|L_1^{\mathscr{D}}| = |\overleftarrow{q3}| = 4$ (based on words aaa, ab, ba, bb), and thus $|L_1^{C_j}| = |L|/\min_{q\in\mathscr{D}} - |L_1^{\mathscr{D}}| = 16/3 - 4 = 1.33$.[9] Sizes of left languages for states $q1$, $q2$, and $q6$ are 1, and the prefix sums equal then $S = \{1, 2, 3\}$, which means that the value of c returned from COMPLOWBOUND is 2. Consequently, we do not need to analyze single state subsets of $\mathcal{Q}$. The final decomposition set for $L = L_1L_2$ is $\{q1, q2, q3, q6\}$ where $L_1 = \{a, aa, aaa, ab, b, ba, bb\}$ and $L_2 = \{\lambda, ab, b\}$.

The fourth refinement makes the algorithm adaptive. We have found out that depending on the input language, the time to execute *Phase1* of basic algorithm could be much longer than the time to execute *Phase2*. This is a disadvantage as in *Phase1* the processes run the same code while in *Phase2* they work in parallel verifying sets

[9] Note that on input to COMPLOWBOUND $D = \mathscr{D}$.

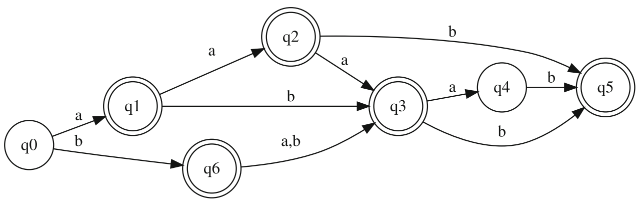

Fig. 12 Transition diagram of DFA accepting language L from Example 9

$\mathscr{D} \cup C_j$. Therefore, when the run time of *Phase2* is shorter compared to *Phase1*, the capacity to take advantage of parallel computation is not fully utilized.

The purpose of *Phase1* is to reduce the set $\mathscr{Q} = \text{STATES}(W) - \mathscr{D}$ so that its size becomes smaller than the threshold T. Apparently, the cause of a long run time of *Phase1* is that the size of $\mathscr{Q}$ remains constant through a series of recursive runs of *Phase1*. So instead of repeating *Phase1*, it is better to start *Phase2* by increasing the value of threshold T. Setting the new value of T (procedure ADJUSTT, Fig. 13) is triggered when a specified number of recursive runs of *Phase1* is completed with no change of $\mathscr{Q}$. More precisely, when the number of runs wherein $old\_s = s$ reaches the fixed value of e, the value of T is increased. However, the rate of growth of T should be controlled so that it does not become too large. Once the value of T is doubled (or tripled) in relation to T_0, the number of recursive runs e to be performed before T is increased again, is also doubled (or quadrupled).

Note that the greater value of threshold T causes the size of set $\mathscr{Q}$ to grow. Consequently, the number of subsets C_j where $C_j \subseteq \mathscr{Q}$, and thus the number of sets $\mathscr{D} \cup C_j$ to verify, increases (the number of subsets C_j is exponential and equal to $2^{|\mathscr{Q}|}$, as the subsets are members of the power set of $\mathscr{Q}$). This means that the degree of parallelism grows, which is desirable since we may use more processes to conduct the search.

The average time complexity of adaptive algorithm given by

$$\mathscr{T}_a(\pi, n) = d(n, \widehat{T}) \cdot (t_1(n, \widehat{T}) + (t_2(n, \varepsilon, \psi\widehat{T}) \cdot \varepsilon \cdot 2^{\psi\widehat{T}})/\pi) \quad (10)$$

this is similar to that of basic algorithm (3). There are, however, two differences. First, the coefficient ε takes into account the fraction of sets $\mathscr{D} \cup C_j$ that skip verification in *Phase2*. Second, the threshold T that was kept constant in the basic algorithm can now be increased adaptively, so it holds that $\widehat{T} \geq T$. The coefficient ε where $\varepsilon \in (0.0, 1.0]$ can considerably reduce the total average run time $d(n, \widehat{T}) \cdot (t_2(n, \varepsilon, \psi\widehat{T}) \cdot \varepsilon \cdot 2^{\psi\widehat{T}})/\pi$ of *Phase2* (see Table 8). Similarly, the growing average value of $\widehat{T}$ resulting from adaptation reduces the amount of computation in *Phase1* while increasing the amount of computation in *Phase2*, which is distributed among π processes.

To conclude, the adaptive algorithm introduces the three refinements aimed at pruning the search space. In contrast to the basic algorithm, which only eliminates particular redundant states, the adaptive algorithm also targets whole sets of states that may generate resultant decompositions. The fourth refinement involving the adjustment of threshold T ensures not only a better balance between the run times of *Phase1* and *Phase2*, but it also provides a better exploitation of parallelism in the decomposition problem. However, all the refinements do not reduce the order of the adaptive algorithm complexity, which remains exponential.

6 Computational experiments

This section reports on the comprehensive experiments conducted to evaluate the performance of basic and adaptive algorithms. The run times to solve the decomposition problem were measured for almost 1450 languages over an alphabet of size $|\Sigma| = 3$–5, and for more than 2700 languages over binary and unary alphabets, and over an alphabet of size $|\Sigma| = 10$ (in what follows we refer to these

Fig. 13 Threshold adjustment procedure

```
1:  procedure ADJUSTT(in $s$)                    ▷ $s$ equals size of set $\mathscr{Q}$ = STATES $(W) - D$
2:      if $old\_s = s$ then
3:          $r \leftarrow r + 1$                 ▷ increase counter $r$ of successive runs wherein $old\_s = s$
4:          if $r = e$ then                      ▷ expected number $e$ of successive runs reached
5:              $T \leftarrow T + 1$; $r \leftarrow 0$     ▷ increase threshold $T$, and reset counter $r$
6:          end if
7:      else
8:          $old\_s \leftarrow s$; $r \leftarrow 0$        ▷ store $s$, and reset counter $r$
9:      end if
10:     if $T \in [2T_0 .. 3T_0)$ then               ▷ increase $e$ to slow down growth of $T$
11:         Keep value of $e$ equal to $2e_0$
12:     else
13:         if $T \geq 3T_0$ then                ▷ increase $e$ again to slow down growth of $T$
14:             Keep value of $e$ equal to $4e_0$
15:         end if
16:     end if
17: end procedure
```

Table 2 Characteristics of languages (a) and automata built by basic (b) and adaptive (c) algorithm (N – number of languages in the set, $|\Sigma|$ – alphabet size; Min, Max – minimum and maximum values, 1st-q, 3rd-q – first and third quartiles, Med (or 2nd quartile) – median; $|L|$ – number of words in the language, $|Q|$ and $|Q'|$ – numbers of states searched by algorithms)

(a)							
Set	N	$\|\Sigma\|$	Min $\|L\|$	1st-q $\|L\|$	Med $\|L\|$	3rd-q $\|L\|$	Max $\|L\|$
E_1	375	3–4	6012	7552	9664	12339	14984
E_2	348	5	60217	66231	74068	81790	89812
P_1	375	3–4	6012	7552	9664	12339	14984
P_2	348	5	60217	66231	74068	81790	89812
(b)							
Set	N	$\|\Sigma\|$	Min $\|Q\|$	1st-q $\|Q\|$	Med $\|Q\|$	3rd-q $\|Q\|$	Max $\|Q\|$
E_1	375	3–4	38	76	89	101	172
E_2	348	5	81	127	148	168	276
P_1	375	3–4	64	97	109	121	192
P_2	348	5	102	152	172	192	295
(c)							
Set	N	$\|\Sigma\|$	Min $\|Q'\|$	1st-q $\|Q'\|$	Med $\|Q'\|$	3rd-q $\|Q'\|$	Max $\|Q'\|$
E_1	375	3–4	35	69	80	92	155
E_2	348	5	66	113	132	153	253
P_1	375	3–4	48	79	90	101	168
P_2	348	5	80	126	144	163	260

alphabets as Σ_{3-5}, Σ_2, Σ_1, and Σ_{10}). Furthermore, the impact of the adaptive setting on the results obtained, and the speed-ups of adaptive algorithm were studied.

The basic and adaptive algorithms[10] were implemented in C language using the MPI library functions in the Intel MPI 5.1.1.109 version. Each process ran a sequential stream of instructions defined by DECOMPOSE (or DECOMPOSE2) procedure. The processes running the algorithms were independent of one another, and synchronized their operation only at the beginning and end of computation. The implementation structure based on the master-worker paradigm is shown in Fig. 14. The aim of the master process was to send the input language L to all the workers, and to collect the decompositions of L found (in the actual implementation, the master process M and worker process W_0 were combined into a single process).

The experiments were carried out on the Tryton supercomputer with a computation speed of 1.48 Pflop/s, running the Linux kernel 2.6.32-754.3.5.el6.x86_64 along with the Slurm utility (Simple Linux utility for resource management). The supercomputer is composed of 1607 compute nodes, each equipped with two 12-core Intel Haswell processors (Xeon E5 v3) operated at 2.3 GHz, with 128 GB of RAM memory. The processors are connected by the 56 Gb/s Infiniband fat tree network. The complete system with a cluster architecture, located in the Computer Centre in Gdańsk, Poland (http://task.gda.pl/centre-en), houses 3214 processors (38568 cores), and 48 Nvidia Tesla accelerators.

[10]The source code of the algorithms and selected benchmark languages used in the experiments are available on the GitHub website and service (https://github.com/tjastrzab/ai).

6.1 Benchmark languages

For the purpose of the experiments, we generated four sets of languages. The sets E_1 and E_2 contained composite languages, while the sets P_1 and P_2 prime languages. The sets E_1 and P_1 included between 6000 and 15000 words, and the sets E_2 and P_2 between 60000 and 90000 words (Table 2a). The composite languages were created using random grammars [17]. Let $\Sigma = \{a_1, a_2, \ldots, a_l\}$ be the set of terminal symbols, $l \geq 1$, let $V = \{V_1, V_2, \ldots, V_r\}$ be the set of nonterminal symbols, $r > l$, and let V_r be the initial symbol. The grammars for composite languages were obtained as follows:

1. For each terminal symbol $a_i \in \Sigma$, create a production $V_i \to a_i$.
2. For each nonterminal symbol V_j where $j = l+1, l+2, \ldots, r-1$:

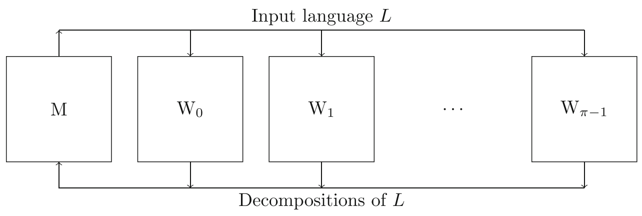

Fig. 14 Master-worker structure of implementation of algorithms (M – master process, W_i, $i = 0, 1, \ldots, \pi - 1$ – worker processes)

- Draw at random a terminal symbol $a \in \Sigma$. Create a production $V_j \rightarrow a$.
- Draw at random l pairs (a, V_i), where $a \in \Sigma$ and $V_i \in V$, $i < j$. Create a production $V_j \rightarrow aV_i$.

3. Create a production $V_r \rightarrow V_{r-2}V_{r-1}$.

When creating the composite languages, we rejected the grammars that generated a number of words outside the ranges 6000–15000 (for set E_1) and 60000–90000 (for set E_2). The values of l and r were selected from the ranges 3–5 and 11–21, respectively, so the maximum length of a word for composite languages was $2(r - l) - 1 = 35$. The sets of prime languages P_1 and P_2 were created based on sets E_1 and E_2. Let $\mathscr{L}$ be a composite language in set E_1 (or E_2). The language $\mathscr{L}$ can be transformed into a prime language $\mathscr{L}'$ belonging to set P_1 (or P_2) using the following steps: (1) find the longest word $\omega \in \mathscr{L}$; (2) generate a random word ω_r over Σ such that $|\omega_r| = |\omega|$; (3) if $\omega_r \notin \mathscr{L}$ then copy language $\mathscr{L}$ into $\mathscr{L}'$, and replace $\omega \in \mathscr{L}'$ with ω_r. (Note that these steps do not guarantee that $\mathscr{L}'$ will always be prime. However, the probability to get a composite language is small. If $\mathscr{L}'$ is composite, one can repeat the steps.)

Consider the size of the input data of the algorithms. There are three independent variables defining this size: the number n of words in L, the size $|\Sigma|$ of the alphabet, and the maximum length h of a word in L. In Section 6.2 we limit the alphabet size to $|\Sigma| = 5$, and the maximum length of a word to $h = 35$. Consequently, the values of $|\Sigma|$ and h become the parameters of the algorithms. So we can assume that the only variable defining the size of the input data of the decomposition problem is the number n of words in L.

6.2 Experimental results

While performing the experiments we ran the basic and adaptive algorithms by employing 16 processes[11] for languages in sets E_1, E_2, P_1, and P_2. We set the maximum run time allowed to solve a given language L to six hours. By *solving* the language we mean that the algorithm either determines all decompositions of L, or finds out that L is prime. Since the basic algorithm failed to solve some languages within the six-hour limit, we defined the *success rate* as

$$R = \frac{N_s}{N} \cdot 100\% \qquad (11)$$

where N_s was the number of languages solved by both algorithms, and N was the number of languages in the set. As shown in Table 3, the adaptive algorithm outperformed the basic algorithm with respect to success rates for all sets under consideration.

A comparison of run times measured by the `MPI_Wtime()` function is shown in Table 4 and Fig. 15. Out of a total of 1446 languages, the comparison relates only to 1168 languages that were solved by both algorithms within the six-hour limit. The box plots of Fig. 15 depict the times through their quartiles. The bottom and top of each box are the first and third quartiles of measurements, and the band inside the box is the second quartile (the median). The lines extending vertically from the boxes, the so-called whiskers, indicate minimum and maximum measurements.

The median run times in Table 4 show that both algorithms solve prime languages faster than composite languages. As the aim is to find all decompositions of a language L, the algorithms have to explore the whole solution space for L. The size of this space is similar for both types of languages, because the cardinalities of languages in sets E_1 and P_1, and E_2 and P_2 are the same. The experiments prove that the number of sets $D \cup C_j$ verified by both algorithms for prime languages is smaller compared to composite languages (Table 5). Consequently, the run times for prime languages are shorter, because fewer decomposition sets need to be verified.

Table 3 Success rates of algorithms

	Basic algorithm			Adaptive algorithm		
Set	N	N_s	R	N	N_s	R
E_1	375	323	86%	375	375	100%
E_2	348	151	43%	348	348	100%
P_1	375	366	98%	375	375	100%
P_2	348	328	94%	348	348	100%
Total	1446	1168		1446	1446	

Table 4 Run times (in seconds) of algorithms (Min, Max – minimum and maximum times, 1st-q, 3rd-q – first and third quartile, Med – median)

Set	N_s	Min	1st-q	Med	3rd-q	Max
	Basic algorithm					
E_1	323	0.4	18.1	98.6	171.4	20948.4
E_2	151	19.2	1028.1	1296.0	1571.2	15403.4
P_1	366	0.3	0.6	1.4	24.6	19619.8
P_2	328	2.0	3.7	5.9	62.3	4132.9
	Adaptive algorithm					
E_1	323	0.2	0.5	0.7	1.2	53.8
E_2	151	4.1	10.7	15.3	22.7	1311.7
P_1	366	0.2	0.3	0.5	0.8	2.7
P_2	328	2.7	7.1	11.2	15.9	56.5

[11] In order to run 16 processes (tasks) on the Tryton cluster, the Slurm utility allocated a single compute node equipped with two 12-core processors, and assigned eight tasks to each processor with one task per core.

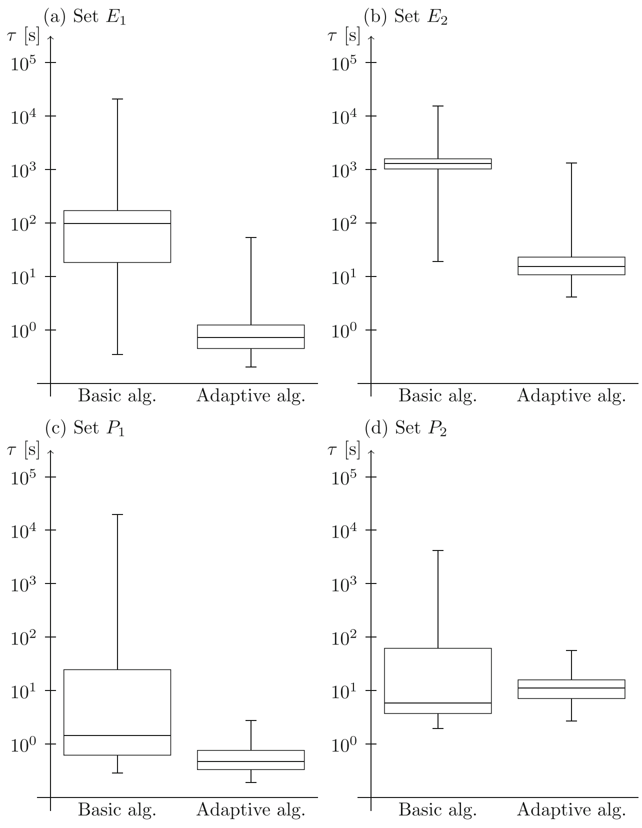

Fig. 15 Comparison of run times (in seconds) of algorithms for 1168 languages

Comparing the median run times (Table 4), we can see that the adaptive algorithm outperforms the basic algorithm for sets E_1, E_2, and P_1. However, for set P_2 the adaptive algorithm shows a slightly worse performance. One of the refinements considers only significant states of automaton A accepting the language L. Due to this refinement, fewer decomposition sets $D \cup C_j$ are verified. The redundant states are removed in the course of building structure W, which is created by procedure BUILDW. Its execution takes a certain amount of time, but one can expect that this amount will be amortized by fewer sets $D \cup C_j$ that need to be verified. However, such amortization did not occur for set P_2, because no sets $D \cup C_j$ for those languages were discovered by the adaptive algorithm (Table 5).

Table 5 Numbers of decomposition sets $D \cup C_j$ verified by the algorithms (IQR – interquartile range, difference between third and first quartile)

		Basic alg.		Adaptive alg.	
Set	N_s	Med	IQR	Med	IQR
E_1	323	$\sim 10^6$	~ 540000	11	45
E_2	151	$\sim 10^6$	~ 16000	7	54
P_1	366	~ 8000	~ 650000	0	0
P_2	328	8	~ 16000	0	0

Table 6 Data samples $d_1 = (n_1, \bar{y}_1, s_1)$ and $d_2 = (n_2, \bar{y}_2, s_2)$ to test the equality of mean run times for the algorithms; the threshold was initially set to $T = 20$ (n_1, n_2 – numbers of measurements; $\bar{y}_1, \bar{y}_2$ – means, in seconds; s_1, s_2 – standard deviations)

	Basic algorithm			Adaptive algorithm			
Set	n_1	$\bar{y}_1$	s_1	n_2	$\bar{y}_2$	s_2	Z
E_1	295	89.4	74.5	345	1.0	0.6	20.4
E_2	115	1230.7	364.8	317	23.6	12.4	35.5
P_1	296	5.3	10.1	341	0.5	0.3	8.2
P_2	280	21.5	32.3	326	11.6	5.6	5.1

Considering the above, we make the claim that the adaptive algorithm is faster than the basic algorithm while solving composite and prime languages. We test this claim statistically on the four pairs of data samples by using the one-sided two-sample test for comparing two means (Table 6). The data samples were created by eliminating the outliers. For example, by means of basic algorithm, the set of 323 measurements for set E_1 was acquired (column N_s in Table 3). From this set, 28 measurements were eliminated as outliers ($n_1 = 295$ in Table 6). A measurement was considered as an outlier if it fell outside the range $[m, M]$ where $m = \text{1st-q} - 1.5 \cdot (\text{3rd-q} - \text{1st-q})$ and $M = \text{3rd-q} + 1.5 \cdot (\text{3rd-q} - \text{1st-q})$ (Table 4).

For samples d_1, d_2, and sets E_i, P_i, $i = 1, 2$, we set up the null and alternative hypotheses

$$H_0 : \mu_1 = \mu_2 \text{ and } H_1 : \mu_1 > \mu_2$$

where μ_1 and μ_2, respectively, are the mean values of run time to solve the population of composite and prime languages by the algorithms. Using the test statistic:

$$Z = \frac{\bar{y}_1 - \bar{y}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

where n_i, $\bar{y}_i$, and s_i are components of sample d_i, the hypothesis H_0 is rejected at the $\alpha = 0.01$ significance level if $Z > z_\alpha$ where $z_\alpha = 2.326$. The sizes of our data samples are in the range of 115–345 (Table 6). In statistics, a sample size of $n_i \geq 30$ is considered large enough to assume that its distribution is normal. Thus, the critical value of z_α is determined based on the standard normal distribution

Table 7 Values of threshold $\widehat{T}$ set by the adaptive algorithm

Set	N_s	Min	Med	Max
E_1	375	20	21	32
E_2	348	20	33	58
P_1	375	20	20	20
P_2	348	20	20	20

Table 8 Variability and average values of terms and coefficients appearing in time complexity formulas given in (3) and (10); times t_1 and t_2 are in seconds

	Basic algorithm ($T = 20, \pi = 16$)			
	E_1		E_2	
Term	Range	Avg	Range	Avg
$d(n, T)$	(2.0, 2.3)	2.1	(2.0, 2.2)	2.1
$t_1(n, T)$	(0.2, 3.1)	1.4	(26.1, 60.5)	45.6
$t_2(n, \psi T)$	(0.001, 0.003)	0.002	(0.01, 0.02)	0.02
ψ	(0.9, 0.9)	0.9	(0.9, 1.0)	1.0
	P_1		P_2	
$d(n, T)$	(1.1, 1.4)	1.3	(1.0, 1.2)	1.1
$t_1(n, T)$	(0.2, 0.8)	0.3	(7.5, 25.6)	14.4
$t_2(n, \psi T)$	(0.0003, 0.001)	0.0005	(0.0005, 0.002)	0.001
ψ	(0.7, 0.9)	0.8	(0.6, 0.8)	0.8
	Adaptive algorithm ($T_0 = 20, \widehat{T} \geq T_0, \pi = 16$)			
	E_1		E_2	
$d(n, \widehat{T})$	(1.0, 1.1)	1.0	(1.0, 1.0)	1.0
$t_1(n, \widehat{T})$	(0.1, 0.7)	0.3	(7.5, 12.8)	9.7
$t_2(n, \epsilon, \psi\widehat{T})$	(0.003, 0.01)	0.004	(0.02, 0.03)	0.02
ψ	(1.0, 1.0)	1.0	(1.0, 1.0)	1.0
ϵ	$(6 \cdot 10^{-7}, 8 \cdot 10^{-5})$	$7 \cdot 10^{-6}$	$(2 \cdot 10^{-12}, 10^{-8})$	$4 \cdot 10^{-12}$
$\widehat{T}$	(21.5, 26.7)	24.7	(36.3, 50.7)	48.0
	P_1		P_2	
	Since no prospective decomposition sets were found, the values of t_1, t_2, ψ, ϵ, and $\widehat{T}$ could not be measured			
Term	Range	Avg	Range	Avg
d	–	0.0	–	0.0

N(0, 1). Clearly, all values of Z are above the critical value of 2.326 (Table 6), so we reject the null hypothesis H_0 in favor of the alternative hypothesis H_1. This means that there is sufficient evidence at the $\alpha = 0.01$ level of significance to claim that the adaptive algorithm solves composite and prime languages faster than the basic algorithm.[12]

Several methods for reducing the search space are proposed in Section 5. The idea behind those methods is to omit the full verification of a set $D \cup C_j$ as to whether it is a decomposition set. When the set is large, its verification can be computationally expensive. However, Lemmas 2-4 allow us to discard a vast majority of the sets without verification. As seen in Table 5, the numbers of sets verified by the adaptive algorithm for composite languages are reduced by several orders of magnitude compared with the corresponding numbers for the basic algorithm. This indicates that the proposed methods of pruning the search space are effective.

[12]This claim is also true if we take into account the outliers. After including the outliers into the data samples d_1 and d_2, all values of Z are still above the critical value. In general, the outlying data points may highly distort the mean and variance of measurements, which, as a result, may give a misleading impression regarding the shape of algorithm run time distributions.

Another way to prune the search space is by removal of redundant states of automaton A accepting the input language L. Due to such removal the state sets of A for the composite languages decrease by approximately 10%, and for the prime languages by approximately 16%–17% (see Med $|Q|$ and Med $|Q'|$ in Table 2b–c). The basic algorithm discovered some prospective decomposition sets for prime languages while the adaptive algorithm did not find any set of that type (Table 5). We believe that the reason for this was smaller automata processed by the adaptive algorithm compared with the basic algorithm.

The adaptive algorithm changes its behavior by setting the value of threshold T (denoted then by $\widehat{T}$) at runtime. The adaptation was most beneficial for languages of set E_2, which turned out to be most demanding in terms of solving the decomposition problem. For these languages the values of $\widehat{T}$ varied in a wide range of 20–58 (Table 7). The capability of adaptation was exploited to a lesser extent for languages in set E_1 with the values of $\widehat{T}$ varying within a range of 20–32, and for the prime languages the adaptive adjustment of $\widehat{T}$ did not occur.

The time complexity formulas $\mathscr{T}_b(\pi, n)$ and $\mathscr{T}_a(\pi, n)$ for the algorithms include several terms and coefficients. To investigate the variability of these quantities, we took the

Table 9 Run times (in seconds) of adaptive algorithm for languages over alphabets Σ_{10}, Σ_2, and Σ_1 where R – range of language size, N – number of languages in the set

Set	R	N	Min	1st-q	Med	3rd-q	Max
Languages over alphabet Σ_{10}							
$\mathscr{E}_1^d$	6–15k	375	0.2	1.4	3.8	5.2	9.6
$\mathscr{E}_2^d$	60–90k	348	2.7	6.6	8.8	11.3	36.8
$\mathscr{P}_1^d$	6–15k	375	0.1	0.2	0.3	0.3	1.3
$\mathscr{P}_2^d$	60–90k	348	1.7	2.9	3.6	5.4	32.1
Binary languages (alphabet Σ_2)							
$\mathscr{E}^b$	6–15k	190	0.5	11.3	75.6	743.8	19574.7
$\mathscr{P}_1^b$	6–15k	375	0.2	0.8	1.0	1.6	6.1
$\mathscr{P}_2^b$	60–90k	348	26.6	50.1	63.2	82.8	253.0
Unary languages (alphabet Σ_1)							
$\mathscr{E}^u$	100–1000	135	55.9	1993.7	5232.4	11807.5	21289.4
$\mathscr{P}^u$	100–1000	220	19.3	811.2	4982.8	11291.9	21311.9

measurements reported in Table 8. The Avg entry contains the average value of a quantity, q, calculated over a language set. The Range entry describes the variability of q through a pair $(q_{\min}, q_{\max})$ where $q_{\min}$ and $q_{\max}$ are the minimum and maximum average values of q calculated over average values in a distinguished interval. The range [6000, 15000] of language size for sets E_1 and P_1 was divided into nine equal intervals, and the range [60000, 90000] for E_2 and P_2 into ten intervals.

The Range values indicate that the times t_1 and t_2 are slowly increasing functions of language size n (Table 8). Recall that t_1 is the average time to find a partial decomposition set D in *Phase1*, and t_2 the average time to verify a set $D \cup C_j$ in *Phase2*. Small values of coefficient ε indicate that pruning of the search space is effective. The values of $d(n, T)$ and $t_1(n, T)$ allow us to estimate the average run time of *Phase1* of basic algorithm for set E_2. This time is $2.1 \cdot 45.6 \approx 95.8$ s. For the adaptive algorithm the run time of *Phase1* is equal to 9.7 s. Since, we have the average run times of complete algorithms ($\bar{y}_1$ and $\bar{y}_2$ for E_2 in Table 6), we can calculate execution times of *Phase2* for both algorithms. We get $1230.7 - 95.8 = 1134.9$ s for the basic algorithm, and $23.6 - 9.7 = 13.9$ s for the adaptive algorithm. Clearly, the run time balance between *Phase1* and *Phase2* is much better for the adaptive algorithm (9.7 vs. 13.9 s) compared to the basic algorithm (95.8 vs. 1134.9 s). The better balance was achieved due to effective pruning of the search space, and to the increase in threshold T done by the adaptive algorithm at runtime.

We also conducted the experiments on languages over a comparatively large alphabet Σ_{10}, and over small alphabets, in particular on the binary and unary languages. The setting was the same as before. The adaptive algorithm was run by 16 processes, and the time limit for solving a language was six hours. The languages over alphabets Σ_2 and Σ_{10}

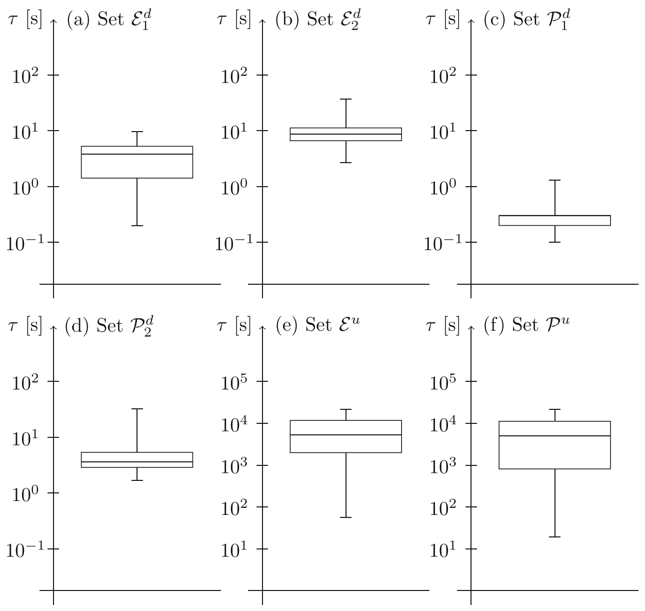

Fig. 16 Run times (in seconds) of adaptive algorithm for languages over alphabet Σ_{10}, and for unary languages

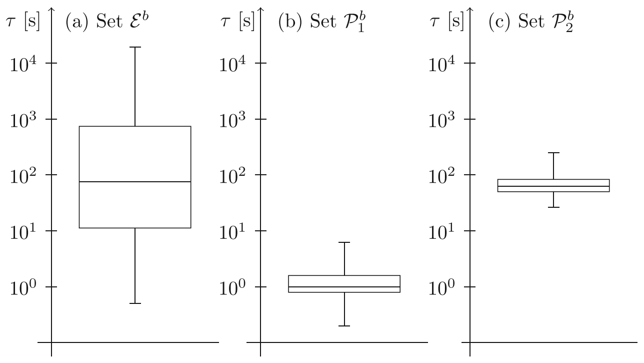

Fig. 17 Run times (in seconds) of adaptive algorithm for binary languages

Table 10 Sizes of state sets Q' of automata searched by adaptive algorithm

Set	N	$\lvert\Sigma\rvert$	Min $\lvert Q'\rvert$	1st-q $\lvert Q'\rvert$	Med $\lvert Q'\rvert$	3rd-q $\lvert Q'\rvert$	Max $\lvert Q'\rvert$
$\mathscr{E}_1^d$	375	10	22	42	50	59	90
$\mathscr{E}_2^d$	348	10	44	84	98	113	166
$\mathscr{P}_1^d$	375	10	29	47	55	64	94
$\mathscr{P}_2^d$	348	10	53	92	106	121	172
$\mathscr{E}^b$	190	2	33	85	104	126	202
$\mathscr{P}_1^b$	375	2	39	108	126	150	248
$\mathscr{P}_2^b$	348	2	149	225	261	317	553
$\mathscr{E}^u$	135	1	1914	1993	1997	1999	2000
$\mathscr{P}^u$	220	1	1905	1987	1994	1997	1999
Total	2714						

were created in a similar fashion as described in Section 6.1. To produce unary languages, the random number of ones forming the words of a language were generated.

The experiments have shown that the languages over an alphabet Σ_{10} were easy to solve. Their decomposition times, in the order of seconds (column Med in Table 9, and Fig. 16a–d), compared favorably with the languages over an alphabet Σ_{3-5} (column Med in Table 4, and Fig. 15). The experiments revealed that the binary languages were harder to solve, while the unary languages were the worst-case input data to solve the problem. The median run times for the binary languages were in the order of tens of seconds, and for the unary languages somewhat longer than 80 minutes (column Med in Table 9, and Figs. 17 and 16e–f). A major difficulty in solving these languages were the large sizes of automata that had to be searched. The ranges of median size of automata accepting the binary and unary languages were, respectively, [104, 261] and [1994, 1997] (column Med$\lvert Q'\rvert$ in Table 10). As a result, the run times for those larger automata were longer, compared to the languages over an alphabet Σ_{10} for which the sizes of automata were in the range of [50, 106].

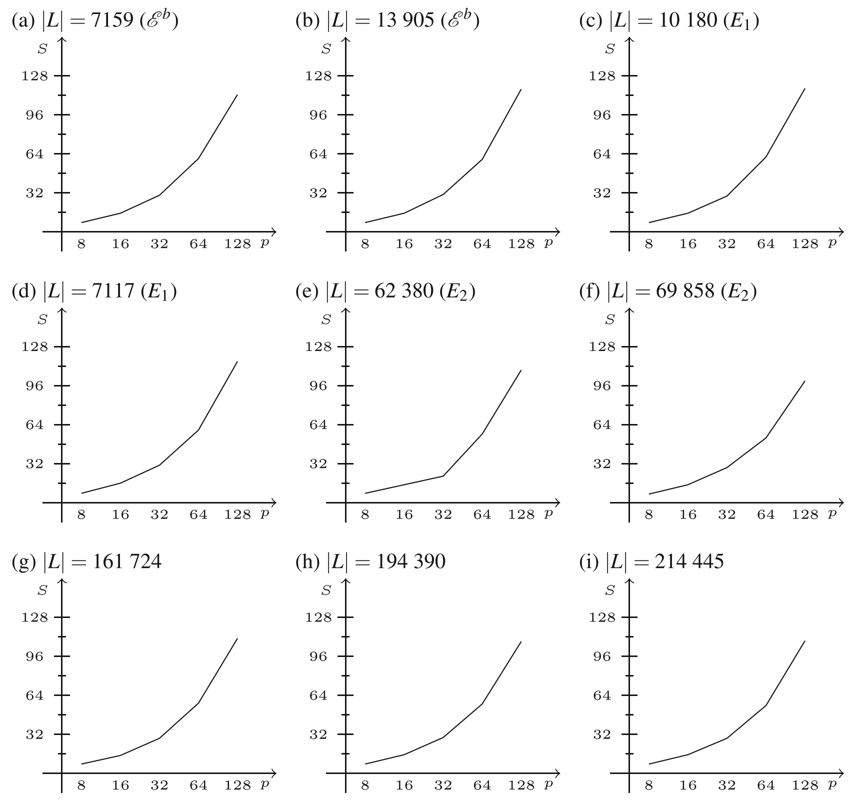

Fig. 18 Speed-ups, S, for binary languages (**a**)–(**b**), and for languages over alphabet Σ_{3-5}: sets E_1 (**c**)–(**d**), E_2 (**e**)–(**f**), and large sets (**g**)–(**i**), ($\lvert L\rvert$ – number of words in the language, p – number of processes)

6.2.1 Scalability study

The results of the adaptive algorithm speed-up evaluation are presented in Fig. 18.[13] The speed-ups achieved for the binary languages (set $\mathscr{E}^b$) and the sets of languages over an alphabet Σ_{3-5} (set E_1) were quite good. For the set E_2 the result was satisfactory.

As can be seen, the speed-ups obtained are not linear. The reason for this is that the parallel processes in the algorithm execute the same code in *Phase1*. So, the overhead of excess computation performed by the processes occurs, which decreases the speed-up. We have significantly reduced that overhead by shortening the run time of *Phase1* (compare Avg times t_1 for the basic and adaptive algorithms in Table 8). It was done by means of the algorithmic refinements, in particular by removing redundant states in procedures RemoveRed2 and BuildW, and by the adaptive adjustment of threshold T.

The large languages in size ranging from 160000 to more than 200000 words scaled very well (Fig. 18g–i). The computational work for these languages was higher, and so the impact of the overhead on the speed-up obtained was smaller. The times to find decompositions of large languages by using 128 processes varied in a range of 7–37 minutes.

As mentioned before, we solve the problem of finding all decompositions of a finite language L in the form of $L = L_1L_2$. The language L does not have to satisfy any specific conditions. To the best of our knowledge, parallel algorithms to solve this problem have not been presented in the literature so far. Therefore we could not compare the outcome of our experiments with the results of other algorithms.

7 Conclusions and future work

In this paper the problem of finite language decomposition is investigated. The problem under consideration, assuming that a language is given as a DFA, is NP-hard. The main contribution of the paper is the adaptive parallel algorithm based on an exhaustive search used for finding all decompositions of a given finite language. The algorithm implements several methods for pruning the search space. Furthermore, the algorithm is adaptive; it modifies its behavior at the time it is run by adjusting one of the parameters based on the runtime acquired data related to its performance. As a consequence, a substantial reduction in the amount of computation necessary to solve the problem has been achieved.

Comprehensive computational experiments carried out on almost 1450 languages over an alphabet Σ_{3-5} proved that the methods for pruning the search space proposed in Lemmas 2–4 were very effective. The methods allowed the adaptive algorithm to reduce the search space by several orders of magnitude compared with the basic algorithm. As a result, the median run time to solve the languages in set E_2 by the adaptive algorithm was equal to approximately 15 s whereas by the basic algorithm it was 1296 s. The adaptive feature of the algorithm proved most beneficial for languages from set E_2 for which the value of threshold T varied in a range of 20–58. The higher value of T is advantageous, because it gives rise to an increase in computational parallelism, which enables better use of available processes.

We also tested more than 2700 languages over a large alphabet Σ_{10} and over small alphabets, specifically the binary and unary languages. The results indicated that the languages over an alphabet Σ_{10} were easier to solve than those over an alphabet Σ_{3-5}. Furthermore, it took longer to decompose the binary languages in comparison to the languages over the alphabets Σ_{10} and Σ_{3-5}, while the unary languages turned out to be the worst-case input data to solve the decomposition problem. Based on these findings, we conclude that finite languages over small alphabets are more difficult to decompose than those over large alphabets.

The scalability study revealed that the binary languages, and the languages generated over an alphabet $|\Sigma| = 3$–5, containing from 6000 to more than 200000 words, scaled well, especially those with larger sizes.

In terms of future work, the two issues can be investigated. The first is adaptive setting of the algorithm, which we believe has the potential to be improved. Presently, the algorithm establishes the value of threshold T based solely on the number of recursive runs of *Phase1*. We suppose that the number of processes executing the algorithm should also be considered while determining the value of T. Another issue to investigate is the further scalability of the adaptive algorithm. At present, the algorithm by using 16 processes can solve the language instances of up to 90000 words in the median run time of tens of seconds, and by using 128 processes the languages of sizes between 160000 and more than 200000 words, in the run time of tens of minutes. The question is to what extent the language size could be enlarged by increasing the number of processes while maintaining the short run time of the algorithm, and possibly high processor utilization.

[13]For the experiments involving 128 processes (tasks), the Slurm utility allocated six compute nodes, and two nodes received 22 tasks, and the other 21 tasks each. Within a node the tasks were assigned as evenly as possible between the two 12-core Haswell processors.

Acknowledgements The research presented in this paper was supported by the National Science Centre Poland (NCN) project 2016/21/B/ST6/02158, and it was co-financed by the Silesian

University of Technology grant for maintaining and developing research potential. The computation of the project was carried out using the infrastructure supported by the Silesian BIO-FARMA project POIG.02.01.00-00-166/08 and POIG.02.03.01-24-099/13 grant GeCONiI. We thank the following computing centres where the computation of our project was also carried out: Academic Computer Center in Gdańsk, Interdisciplinary Centre for Mathematical and Computational Modeling at Warsaw University (computing grant G27-9), and Wrocław Centre for Networking and Supercomputing (computing grant 30).

Funding The research was funded by the National Science Centre Poland (NCN) project 2016/21/B/ST6/02158, and by the Silesian University of Technology grant for maintaining and developing research potential.

Code and data availability The source code of the algorithms and selected benchmark languages used in the experiments are available on the GitHub website and service (https://github.com/tjastrzab/ai).

Declarations

Conflicts of interest The authors declare that they have no conflict of interest.

References

1. Sieder P (2019) A lower bound for primality of finite languages. CoRR arXiv:1902.06253
2. Marin M, Kutsia T (2010) On the computation of quotients and factors of regular languages. Front Comput Sci China 4(2):173–184
3. Afonin S, Golomazov D (2009) Minimal union-free decompositions of regular languages. In: Language and Automata Theory and Applications, Third International Conference, LATA 2009. Proceedings, Tarragona, pp 83–92
4. Wieczorek W, Nowakowski A (2015) Grammatical inference for the construction of opening books. In: 2015 Second International Conference on Computer Science, Computer Engineering, and Social Media (CSCESM). IEEE, pp 19–22
5. Bravo HJ, Pena PN, Alves LVR, Takahashi RHC (2018) Factorization-based approach for computing a minimum makespan controllable sublanguage. IFAC-PapersOnLine 51:19–24
6. Han YS, Salomaa K (2011) Overlap-free languages and solid codes. Int J Found Comput Sci 22(5):1197–1209
7. Hung KV (2013) A prime decomposition algorithm for supercodes. J Comput Sci Cybern 29(4):351–357
8. Czyzowicz J, Fraczak W, Pelc A, Rytter W (2003) Linear-time prime decomposition of regular prefix codes. Int J Found Comput Sci 14(6):1019–1032
9. Han Y-S, Wang Y, Wood D (2006) Infix-free regular expressions and languages. Int J Found Comput Sci 17(2):379–393
10. Han Y-S, Wood D (2007) Outfix-free regular languages and prime outfix-free decomposition. Fund Inf 81(4):441–457
11. Domaratzki M, Salomaa K (2011) On language decompositions and primality. In: Calude CS, Rozenberg G, Salomaa A (eds) Language and Automata Theory and Applications, LNCS, vol 6570. Springer, Berlin, pp 63–75
12. Salomaa A, Salomaa K, Yu S (2008) Length codes, products of languages and primality. In: Martin-Vide C, Otto F, Fernau H (eds) Language and Automata Theory and Applications, LNCS, vol 5196. Springer, Berlin, pp 476–486
13. Han Y-S, Salomaa A, Salomaa K, Wood D, Yu S (2007) On the existence of prime decompositions. Theor Comput Sci 376(1-2):60–69
14. Mateescu A, Salomaa A, Yu S (2002) Factorizations of languages and commutativity conditions. Acta Cybern 15:339–351
15. Salomaa A, Yu S (2000) On the decomposition of finite languages. In: Rozenberg G, Thomas W (eds) Developments in Language Theory: Foundations, Applications and Perspectives. World Scientific Publishing, Singapore, pp 22–31
16. Mateescu A, Salomaa A, Yu S (1998) On the decomposition of finite languages. Turku Centre for Computer Science
17. Wieczorek W (2009) An algorithm for the decomposition of finite languages. Log J IGPL 18(3):355–366
18. Wieczorek W (2009) Metaheuristics for the decomposition of finite languages. In: Kłopotek MA, Przepiórkowski A, Wierzchoń ST, Trojanowski K (eds) Recent Advances in Intelligent Information Systems. Akademicka Oficyna Wydawnicza EXIT, Warszawa, pp 495–505
19. Jastrząb T, Czech ZJ (2014) A parallel algorithm for the decomposition of finite languages. Studia Informatica 35(4):5–16
20. Jastrząb T, Czech ZJ, Wieczorek W (2015) A parallel algorithm for decomposition of finite languages. In: Joubert GR, Leather H, Parsons M, Peters FJ, Sawyer M (eds) Parallel Computing: On the Road to Exascale. IOS Press, Netherlands, pp 401–410
21. de la Higuera C (2010) Grammatical inference: Learning automata and grammars. Cambridge University Press, New York
22. Hopcroft JE, Motwani R, Ullman JD (2013) Introduction to automata theory, languages, and computation, 3rd ed. Pearson international, Addison-Wesley
23. Martens W, Niewerth M, Schwentick T (2010) Schema design for XML repositories: complexity and tractability. In: Paredaens J, van Gucht D (eds) Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. ACM, New York, pp 239–250
24. Niewerth M (2015) Data definition language for xml repository management systems. Ph.D. Thesis, Technischen Universitat Dortmund an der Fakultat Informatik
25. Wieczorek W (2017) Grammatical inference: Algorithms, routines and applications, vol 673. Studies in Computational Intelligence. Springer
26. Wieczorek W (2010) A local search algorithm for grammatical inference. In: Sempere JM, García P (eds) Grammatical Inference: Theoretical Results and Applications, LNCS, vol 6339. Springer, Berlin, pp 217–229
27. Wieczorek W (2016) Inductive synthesis of cover-grammars with the help of ant colony optimization. Found Comput Decis Sci 41(4):297–315
28. Runge T, Schaefer I, Cleophas L, Watson BW (2017) Many-MADFAct: Concurrently constructing MADFAs. In: Holub J, Zdárek J (eds) Proceedings of the Prague Stringology Conference 2017, Prague, pp 126–142

Tomasz Jastrząb received his MSc (2011) and PhD (2019) in Computer Science from the Silesian University of Technology, Gliwice, Poland. His research interests focus on parallel computing, natural language 4 processing, and finite automata. He is also a Java Developer and Team Leader at Technicenter Sp. z o.o., where he works on interdisciplinary projects related to economics, neuropsychology, and computer science.

Zbigniew J. Czech is a Professor of Computer Science at the Silesian University of Technology, Gliwice, Poland. His research interests include computer programming, design and analysis of algorithms, and parallel computing, on which he has more than 45 years of experience lecturing and conducting research. He has served as a research fellow at the University of York and the University of Canterbury in the United Kingdom, and has lectured at numerous universities in Poland and elsewhere, including the University of California–Santa Barbara, Indiana University-Purdue University, and the University of Queensland.

Wojciech Wieczorek received his PhD (2004) and DSc (habilitation degree, 2018) from the Faculty of Automatic Control, Electronics and Computer Science at the Silesian University of Technology, Gliwice, Poland. His research interests include algorithms and data structures, combinatorial optimization, heuristic methods, and answer set programming. Currently, he is a university professor at the University of Bielsko-Biala, Department of Computer Science and Automatics, on a full-time contract. The university is located in the southern, mountainous part of Poland.