A Simple and Optimal Complementation Algorithm for Büchi Automata

Joel D. Allred
Diffblue Ltd
Oxford, UK
joel.allred@diffblue.com

Ulrich Ultes-Nitsche
University of Fribourg
Fribourg, Switzerland
uun@unifr.ch

Abstract

Complementation of Büchi automata is complex as Büchi automata in general are nondeterministic. A worst-case state-space growth of $O((0.76n)^n)$ cannot be avoided. Experiments suggest that complementation algorithms perform better on average when they are structurally simple. We present a simple algorithm for complementing Büchi automata, operating directly on subsets of states, structured into state-set tuples (similar to slices), and producing a deterministic automaton. Then a complementation procedure is applied that resembles the straightforward complementation algorithm for deterministic Büchi automata, the latter algorithm actually being a special case of our construction. Finally, we prove our construction to be optimal, i.e. having an upper bound in $O((0.76n)^n)$, and furthermore calculate the 0.76 factor in a novel exact way.

Keywords Büchi Automaton, Complementation, Optimal Upper Bound, Benchmarking

ACM Reference Format:
Joel D. Allred and Ulrich Ultes-Nitsche. 2018. A Simple and Optimal Complementation Algorithm for Büchi Automata. In *LICS '18: LICS '18: 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, July 9–12, 2018, Oxford, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3209108.3209138

1 Introduction

Checking ω-language containment is important in linear-time temporal verification. Testing $B \subseteq P$ is done algorithmically by testing $B \cap \overline{P} = \emptyset$, where $\overline{P}$ is the complement of P. If P is regular, a Büchi-automaton representing $\overline{P}$ can be constructed effectively. In the worst case, the automaton for $\overline{P}$ can have $O((0.76n)^n)$ many states [10, 17], where n is the number of states of the automaton for P.

Complementation of Büchi automata is difficult because in general, Büchi automata cannot be made deterministic. Complementation of a deterministic Büchi automaton A is, however, quite simple: construct a copy of A, remove all accepting states in the copy, make the remaining states in the copy accepting, make all states in A non-accepting, and finally allow nondeterministically moving from A to corresponding states in its copy. So the complement automaton to a deterministic Büchi automaton consists of two deterministic automata (let's call them an upper and a lower automaton) with transitions from upper to the lower automaton but not back. We adapt this construction to nondeterministic automata by constructing a deterministic version of a given automaton and a decorated copy of the deterministic automaton that deals with acceptance. The result is again an automaton consisting of two deterministic automata that are then combined by allowing to move nondeterministically from the upper to the lower one but not vice versa.

The run analysis that we developed independently leads to similar results as the analysis by Fogarty, Kupferman, Vardi, and Wilke [3] who translate the slice-based approach in [5] to a rank-based approach. Instead of unifiying interim constructions, we directly construct a complement automaton using the sole analysis of the runs of the original automaton, in a similar way deterministic Büchi automata are complemented. Further comparison with existing work can be found in Section 5. Furthermore, the "subset-tuple" construction that we develop leads to similarly structured states like the construction by Fisman and Lustig from nondeterministic Büchi to deterministic parity automata [2].

In our construction, the key notion will be that of a *greedy* accepting run. In short, a greedy run is a run that always aims at reaching an accepting state as quickly as possible. Our construction will consider only greedy runs. By doing so, visits of sets of accepting states in runs of the constructed automaton allow faithfully to identify whether or not they can also occur in a (greedy) run of the given automaton. We basically conduct a subset construction to which we add sufficient structure to identify greedy runs: the states are tuples of sets of states of the given automaton, and sets of non-accepting states and sets of accepting states are never mixed (to avoid losing too much information [15]). The complementation procedure for deterministic Büchi automata [7] turns out to be a special case of our construction.

2 Preliminaries

Let Σ be a finite set. Then Σ^* is the set of all finite words and Σ^ω is the set of all infinite words (ω-words) over Σ. A (finitary) language is a subset of Σ^* and an ω-language is a subset of Σ^ω. We use a functional notation: for (ω-)word x, $x(i)$ is the $(i+1)$st symbol in x, and $x(i,j)$ is the subword $x(i)x(i+1)\cdots x(j-1)$. $|x|$ is the length of x.

In fact, we will overload the "$|\cdots|$" notation: if x is a set, then $|x|$ will be x's cardinality (number of elements), and if x is a tuple, then $|x|$ will be x's size (number of components of the tuple).

Büchi automaton $A = (Q, \Sigma, \delta, q_{in}, F)$ consists of finite state set Q, alphabet Σ, transition function $\delta : Q \times \Sigma \to 2^Q$, initial state $q_{in} \in Q$, and set $F \subseteq Q$ of accepting states. δ is extended in the usual way to finite words and state sets.

Let $x \in \Sigma^\omega$. A run r of A on x is an ω-word of states such that $r(0) = q_{in}$ and $r(i+1) \in \delta(r(i), x(i))$, for all $i \geq 0$. Runs on finite words are defined similarly.

LICS '18, July 9–12, 2018, Oxford, United Kingdom

ACM ISBN 978-1-4503-5583-4/18/07…$15.00
https://doi.org/10.1145/3209108.3209138

Let $\omega(r)$ be the set of all states that recur infinitely often in r. Run r is accepting if and only if $\omega(r) \cap F \neq \emptyset$. Automaton A Büchi-accepts x if and only if there exists an accepting run of A on x [1, 12]. The set of all ω-words accepted by A is the ω-language that A accepts.

If, for all $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is at most a singleton set, then A is deterministic. Otherwise it is nondeterministic. For ω-languages, the expressiveness of deterministic and nondeterministic automata differ: There exist ω-languages that require that each automaton Büchi-accepting them is nondeterministic [1, 12].

We call a (finite) run r on a finite word *greedy* if it always reaches accepting states as quickly as possible. One way of defining greediness is with the help of a function $\alpha : Q \to \{0, 1\}$ that assigns 1 to accepting states and 0 to non-accepting ones, and extend it to sequences of states by its stepwise application. For a run r, we interpret $\alpha(r)$ as binary numbers with the most significant bit to the left. Run r on finite word w is *greedy* if and only if there does not exist a finite run r' on w with $r'(0) = r(0)$ and $r'(|r|-1) = r(|r|-1)$ such that $\alpha(r) < \alpha(r')$ ("<" is the ordering relation on (binary) numbers).

Infinite run r on ω-word x is greedy if and only if all of its finite prefixes $r(0, n+1)$ are greedy on $x(0, n)$.

For automaton A and ω-word x, we can show:[1]

Lemma 2.1. *There exists an accepting run of A on x if and only if there exists a greedy accepting run of A on x.*

3 The Complementation Construction

We claim that our construction is *simple*. In upcoming subsections, we give the formal definition of the complement automaton, which may not look that simple at first glance. Therefore we would like to give here the idea underlying our construction.

We nearly apply the standard subset construction to determinize automata, we just add a little structure to the state sets: we partition them into sets of states of the same *level of greediness* and sort the partition accordingly. The greedier a run to a state, the more to the right does it appear in our construction. Applying that reasoning leads to the construction in Section 3.1 (we call the resulting automaton the *upper automaton*).

According to Lemma 2.1, if there is no greedy accepting run, there is no accepting run on an ω-word. So the complement automaton must accept all ω-words for which all greedy runs visit accepting states only finitely often. The *upper automaton* is a deterministic representation of all greedy runs. In Section 3.3, we therefore create an annotated (colored) copy of the *upper automaton* (we call the resulting automaton the *lower automaton*), in which the coloring guarantees that only ω-words are accepted for which greedy runs do not visit accepting states. We require the *upper automaton* to jump nondeterministically (after at most finitely many visits of accepting states in a greedy run) to the *lower automaton* (no visits of accepting states in greedy runs — visits of accepting states must be *discontinued*, i.e. they are in non-greedy runs). The resulting automaton will accept all ω-words for which greedy runs of the given automaton contain only finitely many accepting states, i.e. it accepts exactly all ω-words not accepted by the given automaton.

[1]Even though we developed the concept of greediness independently, it turns out to be similar to the lexicographical ordering introduced in [3]. The subsequent lemma is then an adapted version of Lemma 6 in [3].

3.1 Determinizing the Automaton

Let $A = (Q, \Sigma, \delta, q_{in}, F)$ be the Büchi automaton we want to complement. We construct an interim deterministic Büchi-automaton $A' = (Q', \Sigma, \delta', (\{q_{in}\}), \emptyset)$ from which we then derive the complement automaton. The state set of A' contains non-empty disjoint sets of A-states:[2]

$$Q' = \bigcup_{m=1}^{|Q|} \left\{(S_1, \ldots, S_m) \in (2^Q \setminus \{\emptyset\})^m \mid (\forall j \neq k : S_j \cap S_k = \emptyset)\right\}.$$

A' does not contain accepting states. Now we define transition function $\delta' : Q' \times \Sigma \to Q'$. Let $p' \in Q'$ and let $m = |p'|$. Let $p'(j)$ be the jth component of p' ($p'(j)$ is a set of A-states). For $a \in \Sigma$, we define the a-successor of the jth component of p' to be:

$$\sigma(p', j, a) = \delta(p'(j), a) \setminus \bigcup_{k=j+1}^{m} \delta(p'(k), a).$$

$\sigma(p', j, a)$ contains all A-states that are a-successors of A-states in $p'(j)$ and not already contained in $\sigma(p', k, a)$, for $k > j$. From this, we get immediately that sets $\sigma(p', j, a)$ and $\sigma(p', k, a)$ are disjoint.

The definition of $\sigma(p', j, a)$ guarantees that if an A-state could occur in multiple components of a tuple, we will keep it only in the rightmost component. So even though $\delta(p'(j), a)$ may not be empty, $\sigma(p', j, a)$ still can be. We partition each set $\sigma(p', j, a)$ into non-accepting A-states and accepting A-states:

$$\sigma_n(p', j, a) = \sigma(p', j, a) \cap (Q \setminus F),$$
$$\sigma_a(p', j, a) = \sigma(p', j, a) \cap F.$$

The resulting sets are still pairwise disjoint. We put $\sigma_a(p', j, a)$ to the right of $\sigma_n(p', j, a)$ and remove all empty sets. We define the transition function of A':

$$\delta'(p', a) = q',$$

where q' is obtained by removing all empty sets in

$$\big(\sigma_n(p', 1, a), \sigma_a(p', 1, a), \ldots, \sigma_n(p', m, a), \sigma_a(p', m, a)\big)$$

but otherwise keeping the order of the non-empty components.

Lemma 3.1. *For all reachable states $p' \in Q'$ in A' and all j, k with $1 \leq j < k \leq |p'|$, $p'(j)$ and $p'(k)$ are nonempty and disjoint.*

3.2 An Example

We take the automaton A in Figure 1 as an example that Büchi-accepts all ω-words over $\{a, b\}$ that contain finitely many occurrences of symbol a, and construct A' stepwise from it.

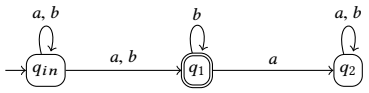

Figure 1. Büchi automaton A accepting $\{a, b\}^* \cdot \{b\}^\omega$.

The initial state of A' is the 1-tuple containing A-state set $\{q_{in}\}$ (see Figure 2). With respect to A's transition relation, we get

$$\sigma((\{q_{in}\}), 1, a) = \delta(\{q_{in}\}, a) = \{q_{in}, q_1\},$$
$$\sigma((\{q_{in}\}), 1, b) = \delta(\{q_{in}\}, b) = \{q_{in}, q_1\}.$$

[2]By "A-states" we refer to the states of automaton A.

Because $\{q_{in}, q_1\}$ contains non-accepting A-state q_{in} as well as accepting A-state q_1, $\{q_{in}, q_1\}$ is partitioned into two sets, and we get $(\{q_{in}\}, \{q_1\})$ as the successor of $(\{q_{in}\})$ when A' reads a or b (see Figure 2). In the next step, we calculate for symbol a:

$$\sigma((\{q_{in}\}, \{q_1\}), 2, a) = \delta(\{q_1\}, a) = \{q_2\}$$

and

$$\begin{aligned}\sigma((\{q_{in}\}, \{q_1\}), 1, a) &= \delta(\{q_{in}\}, a) \setminus \delta(\{q_1\}, a)\\ &= \{q_{in}, q_1\} \setminus \{q_2\} = \{q_{in}, q_1\}.\end{aligned}$$

As in the previous step, $\{q_{in}, q_1\}$ is partitioned into the two sets $\{q_{in}\}$ and $\{q_1\}$, and we get $(\{q_{in}\}, \{q_1\}, \{q_2\})$ as the a-successor of $(\{q_{in}\}, \{q_1\})$ (as can be seen in Figure 2). Similarly, for symbol b, we calculate:

$$\sigma((\{q_{in}\}, \{q_1\}), 2, b) = \delta(\{q_1\}, b) = \{q_1\}$$

and

$$\begin{aligned}\sigma((\{q_{in}\}, \{q_1\}), 1, b) &= \delta(\{q_{in}\}, b) \setminus \delta(\{q_1\}, b)\\ &= \{q_{in}, q_1\} \setminus \{q_1\} = \{q_{in}\}.\end{aligned}$$

No sets need to be partitioned and we get $(\{q_{in}\}, \{q_1\})$ as the b-successor of $(\{q_{in}\}, \{q_1\})$ (see again Figure 2). Now we calculate for symbol a:

$$\begin{aligned}\sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 3, a) &= \delta(\{q_2\}, a) = \{q_2\},\\ \sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 2, a) &= \delta(\{q_1\}, a) \setminus \delta(\{q_2\}, a)\\ &= \{q_2\} \setminus \{q_2\} = \emptyset,\end{aligned}$$

and

$$\begin{aligned}&\sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 1, a)\\ &= \delta(\{q_{in}\}, a) \setminus (\delta(\{q_1\}, a) \cup \delta(\{q_2\}, a))\\ &= \{q_{in}, q_1\} \setminus \{q_2\} = \{q_{in}, q_1\}.\end{aligned}$$

As previously, $\{q_{in}, q_1\}$ is partitioned into the two sets $\{q_{in}\}$ and $\{q_1\}$, the empty set is removed, and we get that $(\{q_{in}\}, \{q_1\}, \{q_2\})$ is an a-successor of itself. Similarly, for symbol b, we calculate

$$\begin{aligned}\sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 3, b) &= \{q_2\},\\ \sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 2, b) &= \{q_1\},\\ \sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 1, b) &= \{q_{in}\}.\end{aligned}$$

No sets need to be partitioned and $(\{q_{in}\}, \{q_1\}, \{q_2\})$ is also a b-successor of itself, completing the construction. The result of the construction is given in Figure 2.

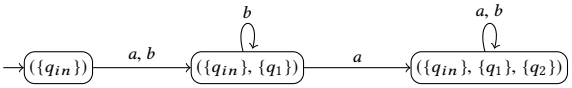

Figure 2. Deterministic automaton A' to A of Figure 1.

3.3 Additional Notation

As pointed out in the previous section, it is important to identify for transitions in A' which component in a successor state results from which component in the predecessor state. In addition, it will be important to identify which component will eventually have no successor component any more (the component disappears, and we will call eventually disappearing components *discontinued*).

Let a be a symbol in Σ. Let p' and q' be two A'-states such that $\delta'(p', a) = q'$. Let $1 \le j \le |p'|$ and $1 \le k \le |q'|$.

If $q'(k) \subseteq \sigma(p', j, a)$, then we write:

$$p'(j) \overset{a}{\mapsto} q'(k),$$

indicating that p''s a-successor q' contains component k *because* p' contains component j. We extend this definition to finite words $w = a_0 a_1 \dots a_l \in \Sigma^*$ in the usual way:

$$q'_0(j_0) \overset{w}{\mapsto} q'_{l+1}(j_{l+1})$$

if and only if there exist states $q'_1, q'_2, \dots, q'_l$ and indices $j_1, j_2, \dots, j_l$ such that for all i, $0 \le i \le l$, we have $q'_i(j_i) \overset{a_i}{\mapsto} q'_{i+1}(j_{i+1})$.

If we take the example from Figure 2, we have, for instance,

$$(\{q_{in}\}, \{q_1\}, \{q_2\})(1) \overset{a}{\mapsto} (\{q_{in}\}, \{q_1\}, \{q_2\})(2),$$

because in the a-transition from state $(\{q_{in}\}, \{q_1\}, \{q_2\})$ to itself, the successor state $(\{q_{in}\}, \{q_1\}, \{q_2\})$ contains component $\{q_1\}$ because $\{q_1\} \subseteq \sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 1, a)$. If we change the symbol, however, then:

$$(\{q_{in}\}, \{q_1\}, \{q_2\})(2) \overset{b}{\mapsto} (\{q_{in}\}, \{q_1\}, \{q_2\})(2),$$

because in the b-transition from state $(\{q_{in}\}, \{q_1\}, \{q_2\})$ to itself, the successor state $(\{q_{in}\}, \{q_1\}, \{q_2\})$ contains component $\{q_1\}$ because $\{q_1\} \subseteq \sigma((\{q_{in}\}, \{q_1\}, \{q_2\}), 2, b)$.

For the remainder of this paper, we always assume that A' is complete. This can be achieved easily by making A complete before constructing A'. It guarantees that for each ω-word x, a unique (A' is deterministic) run of A' on x always exists.

Let $x = x(0)x(1)x(2) \dots \in \Sigma^\omega$. Let $r' = r'(0)r'(1)r'(2) \dots$ be the run of A' on x. Let $i \ge 0$ and let $1 \le j \le |r'(i)|$. We write:

$$r'(i)(j)_\perp$$

to indicate that either $r'(i)(j)$ does not have an $x(i)$-successor (i.e. there does not exist k such that $r'(i)(j) \overset{x(i)}{\mapsto} r'(i+1)(k)$), or for each k such that $r'(i)(j) \overset{x(i)}{\mapsto} r'(i+1)(k)$ we have $r'(i+1)(k)_\perp$. The notation $r'(i)(j)_\perp$ is used to indicate that in the run r' of A' on x, component j of state $r'(i)$ will disappear. Either it disappears immediately (it does not have an $x(i)$-successor) or it disappears eventually (all its $x(i)$-successors will disappear). We say that the component is *discontinued* in A''s run on x. Conversely, we write:

$$r'(i)(j)^\top$$

if and only if $r'(i)(j)_\perp$ does not hold (we then say that $r'(i)(j)$ is *continued* in A''s run on x). In addition, we write $r'(i)(j)^\top_F$ if and only if $r'(i)(j)^\top$ and $r'(i)(j) \subseteq F$, and $r'(i)^\top_F$ if and only if there exists j such that $r'(i)(j)^\top_F$.

If $r'(i)(j)^\top$, then $r'(i)(j)$ has an $x(i)$-successor $r'(i+1)(k)$ such that $r'(i+1)(k)^\top$, because otherwise $r'(i)(j)_\perp$ would hold. Therefore, and because of the pairwise disjointness of components in A'-states (Lemma 3.1), the number of continued components cannot decrease from one state to the next in A''s run on x:

Lemma 3.2. *If k is the number of components $r'(i)(j)$ of state $r'(i)$ such that $r'(i)(j)^\top$ and l is the number of components $r'(i+1)(j)$ of state $r'(i+1)$ such that $r'(i+1)(j)^\top$, then $k \le l$.*

As an example, let $y = bababa \dots$. The run r' of the automaton in Figure 2 on ω-word y is

$$(\{q_{in}\})(\{q_{in}\}, \{q_1\})(\{q_{in}\}, \{q_1\}, \{q_2\})(\{q_{in}\}, \{q_1\}, \{q_2\}) \dots$$

In this run, we label all components of A'-states with $^\top$ or $_\bot$ accordingly, and get:

$$(\{q_{in}\}^\top)(\{q_{in}\}^\top, \{q_1\}^\top)(\{q_{in}\}^\top, \{q_1\}_\bot, \{q_2\}^\top)(\{q_{in}\}^\top, \{q_1\}_\bot, \{q_2\}^\top)\dots$$

The situation changes entirely, when we consider $z = aabbbb\dots$ (two symbols a followed exclusively by infinitely many symbols b). Run r' of A' on z is again:

$$(\{q_{in}\})(\{q_{in}\}, \{q_1\})(\{q_{in}\}, \{q_1\}, \{q_2\})(\{q_{in}\}, \{q_1\}, \{q_2\})\dots$$

However, labeling all components of A'-states with $^\top$ or $_\bot$ leads now to:

$$(\{q_{in}\}^\top)(\{q_{in}\}^\top, \{q_1\}^\top)(\{q_{in}\}^\top, \{q_1\}^\top, \{q_2\}^\top)(\{q_{in}\}^\top, \{q_1\}^\top, \{q_2\}^\top)\dots$$

Because q_1 is an accepting A-state, the run can even be labeled:

$$(\{q_{in}\}^\top)(\{q_{in}\}^\top, \{q_1\}^\top_F)(\{q_{in}\}^\top, \{q_1\}^\top_F, \{q_2\}^\top)(\{q_{in}\}^\top, \{q_1\}^\top_F, \{q_2\}^\top)\dots$$

We discuss in the next section that run r' of A' on ω-word x contains infinitely many states with component labels $^\top_F$ if and only if automaton A Büchi-accepts x.

3.4 Some Properties of the Construction

Let $q' \in Q'$. We will write "$\bigcup q'$" to designate $\bigcup_{i=1}^{|q'|} q'(i)$, i.e. the set of all A-states that occur in components in q'. From the definition of the transition function δ' we get immediately:

Lemma 3.3. $\delta(\bigcup q', a) = \bigcup \delta'(q', a)$.

For the remainder of this section, let $a \in \Sigma$, let $w \in \Sigma^*$, let $x \in \Sigma^\omega$, and let r' be the run of A' on x. It is not difficult to show:

Lemma 3.4. $\delta(\{q_{in}\}, w) = \bigcup \delta'(q'_{in}, w)$.

If we consider in detail what happens in our construction with respect to successor components in A'-states, we can observe that paths of A-states through successor components represent precisely greedy runs of A. Taking into account Lemma 2.1, we can prove:

Lemma 3.5. *A accepts x if and only if r' contains infinitely many A'-states that contain continued sets of accepting A-states (i.e. A'-states of the type $r'(j)^\top_F$, for infinitely many different j).*

In Section 4, we calculate the number of states in A' to be at most $2a(n) - 1$, where n is the number of states in A and $a(n)$ is the nth ordered Bell number. An approximation gives us $O((0.531n)^n)$.

3.5 Complementation

Please recall that A will accept an ω-word x if and only if the run of A' on x contains infinitely many states that contain continued components that are sets of accepting A-states. So the complement automaton A_c that we are going to construct simply must accept all ω-words x such that the run of A' on x visits states with continued accepting components only finitely often.

To achieve this, A_c will be composed of A' and a copy $\check{A}$ of A' in which a coloring of state components will ensure that the visit of a state with a continued accepting component will lead to non-acceptance. By a nondeterministic jump from A' that we call the *upper part* of A_c to a corresponding state in $\check{A}$ that we call the *lower part* we achieve the following: an accepting run of A_c cannot stay in A' as A' does not contain accepting states. So it will have to jump eventually to states in $\check{A}$ and there the appearance and only the appearance of a continued accepting component in a state of the run will prevent $\check{A}$ from accepting. As long as the run of A_c is in A', states with continued accepting components may occur, but only finitely often because of the necessary jump to $\check{A}$ in which states with continued accepting components are then prohibited. So A_c will accept an ω-word x if and only if the run of A' on x would contain only finitely many states with continued accepting components, or in other words: if and only if A does not accept x.

3.5.1 Upper (Non-accepting) Part

We will define a coloring of state components in the lower part to introduce an acceptance condition in the complement automaton. To have a consistent unified notation, we also color each component of the states in A' with color -1, and we call the resulting automaton $\hat{A} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q}_{in}, \hat{F})$.

3.5.2 Lower (Accepting) Part

The lower automaton $\check{A} := (\check{Q}, \Sigma, \check{\delta}, \check{q}_{in}, \check{F})$ can be defined in a similar fashion, but with values $0, 1, 2$ for the coloring indices and a non-empty accepting set $\check{F}$, defined according to the coloring of the components of the states.

In a nutshell, for an ω-word $x \in \Sigma^\omega$ we want the run r_c of A_c on x to be accepting if and only if each greedy run r of the original automaton A on x either:

- eventually stops visiting states of F, or
- is discontinued (i.e. is finite or becomes non-greedy).

We now give some insight into the meaning of the colors. Let i' be the point where run r_c jumps to the lower part, i.e. $i \geq i' \Leftrightarrow r_c(i) \in \check{A}$. The colors of the components of the states of $\check{A}$ hold some information on what happens in the greedy runs of A, "after" i'. For $i \geq i'$, we have:

color $c = 0$: If a tuple component $(S_j, 0)$ is 0-colored in $r_c(i)$, then for each state q in S_j, each greedy run r of A such that $r(i) = q$ has not yet visited an accepting A-state "since" i'.

color $c = 2$: If a tuple component $(S_j, 2)$ is 2-colored in $r_c(i)$, then for each state q in S_j, the greedy run r of A such that $r(i) = q$ has visited an accepting A-state since i'.

color $c = 1$: If a tuple component $(S_j, 1)$ is 1-colored in $r_c(i)$, then for each state q in S_j, the greedy run r of A such that $r(i) = q$ has visited an accepting A-state since i', but there also exist 2-colored components that have not yet disappeared. We say that 1-colored components are *on hold*, meaning that for the moment, we wait until the 2-colored components disappear.

We define $\check{A}$ formally.:

$$\check{Q} := \bigcup_{m=1}^{|Q|} \big\{\big((S_1, c_1), \dots, (S_m, c_m)\big) \in (2^Q \setminus \{\emptyset\} \times [0, 2])^m \mid \forall j \neq k, S_j \cap S_k = \emptyset\big\}.$$

The transition function $\check{\delta} : \check{Q} \times \Sigma \to \check{Q}$ is defined by extending the transition function δ' of Section 3.

We now define the colorings. Let $\check{p} := \big((S_1, c_1), \dots, (S_m, c_m)\big) \in \check{Q}$ be a state of $\check{A}$ and let $p' := (S_1, \dots, S_m)$ be the corresponding state of A' obtained by removing the colors c_j. For $a \in \Sigma$, let $q' = (S'_1, \dots, S'_{m'})$ be the unique state of A' such that $\delta'(p', a) = q'$. We define $\check{\delta}(\check{p}, a) := \check{q}$, for $\check{q} := \big((S'_1, c'_1) \dots, (S'_{m'}, c'_{m'})\big)$ where for each component (S, c) of $\check{p}$ and (S', c') of $\check{q}$ bound by the relation $(S, c) \overset{a}{\mapsto} (S', c')$, the color transitions from c to c' are specified as follows:

- $0 \to 1$: if $S' \subseteq F$ and p' is not a breakpoint,
- $0 \to 2$: if $S' \subseteq F$ and p' is a breakpoint,
- $1 \to 2$: if p' is a breakpoint,

- $c \to c$: otherwise,

where a "breakpoint" is a state of $\check{Q}$ that does not contain a 2-colored component.

Even though it will not be used, we can define $\check{q}_{in} := \big((\{q_{in}\}, 0)\big)$ as the initial state. The set $\check{F}$ of accepting states holds all states that do not contain a 2-colored component :

$$\check{F} := \bigcup \big\{\big((S_1, c_1), \ldots, (S_m, c_m)\big) \in \check{Q} \mid \forall 1 \leq j \leq m, c_j < 2\big\}.$$

3.5.3 The Complement Automaton

We can join $\hat{A}$ and $\check{A}$ to build the complement automaton $A_c = (Q_c, \Sigma, \delta_c, q_c, F_c)$, Büchi-accepting the complement ω-language to the ω-language of the original automaton A:

- $Q_c := \hat{Q} \cup \check{Q}$, $q_c := \hat{q}_{in}$, and $F_c := \check{F}$
- The transitions $\delta_c : Q_c \times \Sigma \to 2^{Q_c}$ permit all transitions within $\hat{A}$ as well as all those within $\check{A}$, plus additional transitions from $\hat{A}$ to $\check{A}$ for all $p = \big((S_1, -1), \ldots, (S_m, -1)\big) \in \hat{A}$ and $a \in \Sigma$:[3]

$$\check{\delta}\Big(\big((S_1, 0), \ldots, (S_m, 0)\big), a\Big) \in \delta_c(p, a)$$

A_c accepts the complement of the ω-language accepted by A. It is worth noting that A_c is deterministic in the limit: after exactly one non-deterministic choice when jumping from the upper to the lower part, A_c behaves deterministically ($\hat{A}$ and $\check{A}$ are deterministic).

Adding the coloring to A' does not introduce more than a factor of 3^i, for each A'-state with i components. A closer analysis yields $O(1.279n)^n$ as an upper bound of the construction (a precise calculation of the upper bound for an improved construction can be found in Section 4.2).

3.6 The Example

We resume our previous example and construct the complement of the automaton of Figure 1.

To avoid cumbersome notation in the following example, a component (S_j, c_j) will be denoted as:

$$\begin{array}{llll} \widehat{S_j} & \text{if } c_j = -1 & \overline{S_j} & \text{if } c_j = 1 \\ S_j & \text{if } c_j = 0 & \overline{\overline{S_j}} & \text{if } c_j = 2 \end{array}$$

The upper part $\hat{A}$ of the complement automaton is just the automaton A' where we append color -1 to all components.

The lower part $\check{A}$ is constructed in a similar way, but with the addition of colors 0, 1 and 2. Let's first look at the a-successor in $\check{A}$ of $(\widehat{\{q_{in}\}})$. The two successor sets are $\{q_{in}\}$ and $\{q_1\}$. By definition of the transition function $\check{\delta}$, since $\{q_{in}\} \cap F = \emptyset$, the color (initially 0) remains 0. For the set $\{q_1\}$, since $\{q_1\} \subseteq F$, the new color is 2. So the a-successor of $(\widehat{\{q_{in}\}})$ in $\check{A}$ is $(\{q_{in}\}, \overline{\overline{\{q_1\}}})$. The same holds for symbol b.

Let's now look at the a-successor of this newly created state. The a-successor of $\{q_1\}$ is $\{q_2\}$ and color 2 remains. The a-successors of $\{q_{in}\}$ are $\{q_{in}\}$ (left successor) and $\{q_1\}$ (right successor). Since $\{q_{in}\} \cap F = \emptyset$, the color of $\{q_{in}\}$ remains 0. The color of $\{q_1\}$ is set to 1 because $\{q_1\} \subseteq F$ and color 2 already exists in the predecessor state. So the a-successor of $(\{q_{in}\}, \overline{\overline{\{q_1\}}})$ is finally $(\{q_{in}\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$.

[3]Transitions from the upper to the lower part are as if originating from entirely 0-colored states in the lower part.

By applying this simple method, we create all reachable A_c-states and get the automaton of Figure 3, where the accepting states are the states of the lower part which do not contain a 2-colored component. Here it is only the case for state $(\{q_{in}\}, \overline{\{q_1\}}, \{q_2\})$.

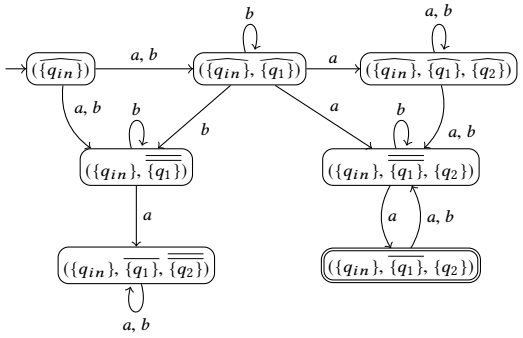

Figure 3. Automaton A_c complement to A.

3.7 Possibilities for Optimization

Without further proof here, it is easy to see that there is some room for optimization. For instance in the case where automaton A is complete, we observe that A_c-states of which the rightmost component has color 2 can be ignored, as well as all their successors. This is because the completeness enforces that state components can only disappear if at some point, the deletion process described in section 3 removes them. Since the deletion process is done right-to-left, this is only possible if there exists another component on the right, which is not the case here. Therefore a rightmost branch is persistent and if its color is 2, this color never disappears and future states can never become accepting.

Taking our previous example, since A is complete, the optimization directly leads to Figure 3 without states $(\{q_{in}\}, \overline{\overline{\{q_1\}}})$ and $(\{q_{in}\}, \overline{\{q_1\}}, \overline{\overline{\{q_2\}}})$ (applied at construction time and not as a reduction from Figure 3).

4 Optimality

A desirable property for a complementation algorithm is that of optimality, i.e. for the maximal state blow-up to be within $O\big((0.76n)^n\big)$. We shall first mention that this 0.76 value is not exact and has been estimated to be approximately 0.7645 using numerical analysis tools (e.g. in [17]). The algorithm presented in Section 3 is not optimal but we give here an alternative formulation that is. Most remarkably, computing the worst-case growth — using a new kind of expression we call the colored ordered Bell numbers — yields an exact representation of the 0.7645 approximation:

$$\frac{1}{\ln(\phi) \cdot e} \approx 0.7645$$

where ϕ is the golden ratio and e is Euler's number.

4.1 Alternate formulation

4.1.1 State merging

The reduction of the state space can be achieved by systematically merging some tuple components together.

We first observe that merging all neighboring 1-colored components, as well as merging neighboring 2-colored components does not alter the resulting language. More precisely, the following replacements can be applied on states of $\check{Q}$:

$$(\ldots,(S_j,1),(S_{j+1},1),\ldots) \to (\ldots,(S_j \cup S_{j+1},1),\ldots)$$
$$(\ldots,(S_j,2),(S_{j+1},2),\ldots) \to (\ldots,(S_j \cup S_{j+1},2),\ldots).$$

Let $join : Q_c \to Q_c$ be the function that operates that joining. Extending that function to sets of states in the usual way (by element-wise application), we define an updated transition function $\delta_m : Q_c \times \Sigma \to 2^{Q_c}$ by $\delta_m(q,a) := join(\delta_c(q,a))$.

Hence from automaton $A_c := (Q_c, \Sigma, \delta_c, q_c, F_c)$ constructed in Section 3, we define automaton $A_m := (join(Q_c), \Sigma, \delta_m, q_c, join(F_c))$ with a reduced state-space. A_m still Büchi-accepts the complement of the original automaton:

Lemma 4.1. *Let r_c be a run of A_c on an ω-word x. Let r_m be a sequence of states such that $r_m(i) = join(r_c(i))$, $\forall i \geq 0$. Then r_m is a run of A_m on x.*

As $r_c(i)$ contains the same 1-colored and 2-colored components as $r_m(i)$, and the fate of these components is bound by the fact that they are neighboring, we get:

Lemma 4.2. *A_c accepts r_c if and only if A_m accepts r_m.*

Building on this, another optimization is to merge any 1-colored component with its preceding neighbor if the latter is a 2-colored component, recursively applying the replacement:[4]

$$(\ldots,(S_j,2),(S_{j+1},1),\ldots) \to (\ldots,(S_j \cup S_{j+1},2),\ldots)$$

We define another function $join_2 : Q_c \to Q_c$ accordingly that, extended to sets in the usual way, leads to a new transition function $\delta_2 : Q_c \times \Sigma \to 2^{Q_c}$ defined by $\delta_2(q,a) := join_2(\delta_m(q,a))$.

The resulting automaton $A_2 := (join_2(Q_c), \Sigma, \delta_2, q_c, join_2(F_c))$ is still a complement automaton:

Lemma 4.3. $L(A_c) = L(A_2)$.

Proof. Let us consider two cases:

1. $L(A_c)$ does not accept the input word.
2. $L(A_c)$ accepts the input word,

Case 1. is easy, because if A_c does not accept a word x, then its run r_c on x must have a continued sequence of 2-colored components. By construction the run r_2 of A_2 on x also contains such a sequence where the only difference is that its components might contain more states than the corresponding components in the sequence in r_c because of the 1-colored components that have been added in. Hence that sequence cannot disappear before the one in r_c. So if r_c does not accept, then r_2 cannot accept either.

For Case 2., we first observe that merging a 1-colored component with a 2-colored component has the effect of pushing back the next breakpoint, because instead of having to wait for the 2-colored components to be discontinued, we also have to wait until the "upgraded" 1-colored components disappear. In order to not delay the next breakpoint indefinitely, we must ensure that such a merging of a 1-colored component with a 2-colored component is only done finitely many times. That is the case when the 1-colored component is to the right of the 2-colored one, because that situation only occurs when a 0-colored component has just disappeared, and since no 0-colored components can appear other than from a pre-existing 0-colored component, only a finite number of "2-1-merges" can occur and the breakpoints are only finitely delayed. □

[4]The operation is applied recurrently as long as the tuple contains a 2-colored component followed by a 1-colored component.

4.1.2 Optimal Version

The optimizations that result in automaton A_2 don't yet achieve a state growth bounded by $O((0.76n)^n)$. To obtain such a result, we must constrain the lower part to have at most one 2-colored component at a time. All other components that would be 2-colored will be 1-colored and be turned 2-colored one after the other in a sort of round robin fashion.

To cater for this we introduce a new interim color 3 to mark the next component that will become 2-colored (thus keeping track of the round robin sequence).

We construct worst-case optimal automaton $\tilde{A} := (\tilde{Q}, \Sigma, \tilde{\delta}, q_c, \tilde{F})$ from A_2. $\tilde{A}$'s states $\tilde{Q}$ are similar to those of A_2, except that in the lower part, states contain at most one component of a color different from 0 and 1, but this other color can be 2 or 3. $\tilde{A}$'s accepting states $\tilde{F}$ are the states of the lower part that do not contain a component of color 2. The transitions $\tilde{\delta} : \tilde{Q} \times \Sigma \to 2^{\tilde{Q}}$ are similar to δ_2 except that the color transition rules from c to c' for components satisfying $(S,c) \overset{a}{\mapsto} (S',c')$ (see Section 3.5.2) now follow:

- $0 \to 1$: if $S' \subseteq F$
- $1 \to 3$: if q' is a breakpoint and if S' is the next 1-colored component to the right of the 0-colored component that follows the position where the disappeared 2-colored component would have been. If the end of the tuple is reached, continue the search from the leftmost component. If there never was a 2-colored component, chose any 1-colored component.
- $3 \to 2$: (always)
- $c \to c$: otherwise.

In other words, the colorings are as in δ_2, with the difference that some 1-colored components are delayed from being switched to 2. To illustrate the round robin, consider the example below (Figure 4) where the 2-colored is discontinued. Since the destination is a breakpoint, we look for the next 0-colored component to the right and then find the next 1-colored component and switch it to color 3.

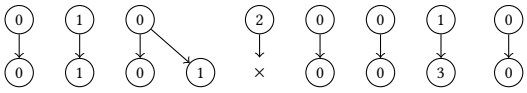

Figure 4. Color transitions of a state of $\tilde{Q}$ to a breakpoint (the numbers depict the component colors).

The intuitive reason for choosing to step over 0-colored states instead of just selecting the next 1-colored component is that it ensures that we do not recurrently select a spawning 1-colored component to be colored 3. Since the number of 0-colored components cannot increase, that forces the round robin to make progress over the tuple and ensures that we ultimately capture a continued sequence of 1-colored components if there is one.

To demonstrate that language acceptance is not altered by these changes, we only discuss the variations in colorings between $L(A_2)$ and $L(\tilde{A})$, as the two constructions are otherwise structurally equivalent and coincide on their upper parts.

Lemma 4.4. $L(A_2) \subseteq L(\tilde{A})$

Proof. To show that $\tilde{A}$ does not under-accept, we must show that an infinite number of breakpoints in A_2 implies an infinite number of breakpoints in $\tilde{A}$. This is fairly straightforward as $\tilde{A}$ "breaks" more often than A_2, because when A_2 needs to "wait" for all its 2-colored components to disappear in order to reach a breakpoint, $\tilde{A}$ will reach a breakpoint whenever it's unique 2-colored component disappears. □

Lemma 4.5. $L(A_2) \supseteq L(\tilde{A})$

Proof. If A_2 does not accept a given word, then it must have an infinite sequence of 2-colored components. Such a sequence also exists in $\tilde{A}$, but it might be partially or totally colored 1 instead of 2 (because in $\tilde{A}$, 2-colorings are "delayed"). If it is only partially 1-colored, it means that under the round robin it eventually reaches color 3, and then immediately 2, and will go on to be an infinite succession of 2-colored components, avoiding any future breakpoint and rendering the run non-accepting. If that sequence remains entirely 1-colored, it means that the round robin never reached that component and thus there must exist another infinite sequence of 2-colored components preventing any further breakpoint from happening. Thus only a finite number of breakpoints are possible and $L(\tilde{A})$ does not accept. □

A_2 and $\tilde{A}$ are hence language equivalent and we finally get:

Corollary 4.6. $\overline{L(A)} = L(\tilde{A}) = L(A_2) = L(A_c)$.

4.2 Worst-case Analysis

Recall that the construction has an "upper" and a "lower" part. To compute the maximum size of the automaton that can be produced, we look at how many different states we can produce in the proposed scheme. We can see that, for the matter of complexity, only the lower part is significant as the number of states of the upper part disappears in the overall measure of complexity. In Section 4.1.2, the state description of the lower part has the following properties (we consider n states in the automaton to complement):

- m states are chosen from the n
- these m states are partitioned into k labeled sets[5]
- each set can be in one of 4 colors within $\{0, 1, 2, 3\}$
- at most one component has a color > 1, which we will call the *marked* component.

To enumerate the possible combinations we start by assigning the marked component, for which there are k positions to choose from. Since that component can be 2- or 3-colored, there are $2k$ possibilities. To take into account the possibility that the marked component is not present, we simply assume that it can be assigned to a component that is 1- or 2-colored[6]. Hence we consider that there are $4k$ ways of assigning the marked component.

We then look at the remaining $k-1$ components which follow these two rules:

- a 0-colored component can be followed by a 0- or a 1-colored component
- a 1-colored component can only be followed by a 0-colored component.

[5]The labeling here expresses the order of the subsets.
[6]This yields an overestimation of the number of combinations which does not affect the final complexity measure.

Let us define $col(t)$ to be the number of possible colorings of a t-tuple in this scheme, and $col_0(t)$, $col_1(t)$ to be the number of colorings of a t-tuple whose leftmost component is colored 0 or 1, respectively. Then the following hold for $k \geq 1$:

- $col(k) = col_0(k) + col_1(k)$
- $col_0(k+1) = col_0(k) + col_1(k)$
- $col_1(k+1) = col_0(k)$

which allows us to give a recursive definition of col $\forall k \geq 1$:

$$col(k+2) = col(k+1) + col(k).$$

Since $col(1) = 2$ (because we can color an unmarked 1-tuple in either color 0 or 1), we have the following alignment on the Fibonacci numbers $\forall k \geq 1$: $col(k) = Fib(k+2)$. It is well known that: $Fib(k) \leq \phi^k$ where $\phi \approx 1.618$ is the golden ratio. Hence the number of colorings of a k-tuple is bounded by $4k\phi^{k+1}$.

In order to get a bound for the size of the lower part, we first consider the upper part which is simpler to compute as it does not involve colorings. The number of weak orderings on a set of j elements (which enumerates the number of states of the upper part that feature j preselected states) is expressed as the j^{th} ordered Bell number:

$$a(j) := \sum_{k=0}^{j} k! \left\{ {j \atop k} \right\}.$$

where $\left\{ {j \atop k} \right\}$ are the Stirling numbers of the second kind:

$$\left\{ {j \atop k} \right\} = \frac{1}{k!} \sum_{t=0}^{k} (-1)^{k-t} \binom{k}{t} t^j.$$

Using an asymptotic approximation of the Bell numbers [4] and Stirling's approximation of the factorial [11] we get the following:

$$a(j) \approx \frac{1}{2(\ln 2)^{j+1}} \cdot j! \approx \frac{\sqrt{2\pi j}}{2 \ln 2} \cdot \left(\frac{j}{\ln 2 \cdot e} \right)^j. \tag{1}$$

Adding the possible ways of selecting these j states within the n states of the original automaton, we get the total number of possible states of the upper part:

$$\#_{upper} = \sum_{j=1}^{n} \binom{n}{j} a(j).$$

Another property of the ordered Bell numbers by Gross [4]:

$$\sum_{j=0}^{n-1} \binom{n}{j} a(n-j) = 2a(n) - 1 \tag{2}$$

gives us a simpler expression of the number of states of the upper part, $\#_{upper}(n) = 2a(n) - 1 \approx 2a(n)$. Approximation (1) leads to:

$$\#_{upper}(n) \approx \frac{\sqrt{2\pi n}}{\ln 2} \cdot \left(\frac{n}{\ln 2 \cdot e} \right)^n.$$

Coming back to the state count for the lower part of the alternate definition, the bound for the colorings of k-tuples yields:

$$\#_{lower}(n) <= \sum_{j=1}^{n} \binom{n}{j} \sum_{k=0}^{j} k! \left\{ {j \atop k} \right\} 4k\phi^{k+1} <= 4n\phi \sum_{j=1}^{n} \binom{n}{j} \sum_{k=0}^{j} k! \left\{ {j \atop k} \right\} \phi^k$$

We introduce the expression appearing above as the nth λ-colored ordered Bell number:

$$a_\lambda(n) := \sum_{j=1}^{n} \binom{n}{j} \sum_{k=0}^{j} k! \left\{ {j \atop k} \right\} \lambda^k.$$

Using techniques inspired by Gross in [4] (but without giving the full proof here) we can show a result that generalizes (2):

$$\sum_{j=1}^{n} \binom{n}{j} a_\lambda(j) = \frac{\lambda+1}{\lambda} \cdot a_\lambda(n) - 1.$$

We can also derive a generalization of the asymptotic approximation:

$$a_\lambda(n) \approx \frac{\sqrt{2\pi n}}{(\lambda+1) \cdot \ln\left(\frac{\lambda+1}{\lambda}\right)} \cdot \left(\frac{1}{\ln\left(\frac{\lambda+1}{\lambda}\right) \cdot e} \cdot n \right)^n$$

leading to a complexity result for the lower part, and thus the general state-growth for the alternate construction[7]:

$$^{\#}_{lower} \in O\Big(\left(\frac{1}{\ln(\phi) \cdot e} \cdot n \right)^n \Big) = O\big((0.7645 \cdot n)^n \big).$$

This shows optimality of our construction by matching the lower bound given in [17] for which we happen to give an exact expression.

5 Comparison to Other Constructions

In this section we go through all major types of complementation constructions and discuss the structural differences with our approach. We then compare efficiency in terms of complement size on random examples. For clarity we refer to the algorithm presented in this paper as the "tuple construction".

5.1 Types of Complementation Algorithms

Existing constructions can generally be divided into four main categories:

- Ramsey-based
- determinization-based
- slice-based
- rank-based.

The obvious example of a Ramsey-based construction is Büchi's complementation method that was introduced in his seminal paper [1]. It uses a Ramsey-type combinatorial argument about equivalence classes of words. Although theoretically interesting, it is not of practical interest as its worst-case growth is $2^{O(n^2)}$.

Determinization-based algorithms essentially operate in three steps. First, the input automaton is converted to an automaton of deterministic kind, such as a Muller or Rabin. The deterministic automaton is complemented, which is usually a trivial operation, and then converted back to Büchi. The best-known determinization algorithm is the one by Safra [9] and has lead to one of the most efficient complementation algorithms, known as Safra-Piterman. Although this method involves an interim automaton that can be very large, experiments show that this can lead to good results, as testified by Tsai et al. in [13].

In slice-based algorithms the constructed state descriptions represent the set of all states that are visited in the input automaton. As such, these states represent *slices* of the run graph of the original automaton. The most representative example of this kind is the one described by Kähler and Wilke in [5]. In that sense, the construction presented in this paper can also be seen as a slice-based construction. Furthermore, the upper part is essentially identical to the non-accepting part of Kähler and Wilke's automaton as the slices are constructed in the same way. The main difference between the two algorithms is that the tuple construction achieves determinism-in-the-limit by enforcing that all branches containing accepting states must eventually die. The slice-based construction by Kähler and Wilke is rather based on guessing which branch will eventually die, thus introducing non-determinism in the infinite part of the constructed automaton.

Rank-based algorithms were devised by Kupferman and Vardi in [6] and are based on the analysis of run graphs in which each node (representing a state of the input automaton) is assigned an integer value called a *rank*. Each run eventually stabilizes in a certain rank. The acceptance condition is based on the parity of these ranks and the following principles are applied:

- at each transition, guess a rank
- keep an obligation set (that holds the nodes that owe a visit to an accepting state) to enforce the parity of the rank.

As we will see in Section 5.2.2, this approach is not very efficient, but is interesting from a theoretical point of view as it was the first construction to have been proven to be optimal.

Although ranks seem to have little to do with the construction presented in this paper, in [3] Fogarty et al. present a way of seeing the weak-orderings of the slice-based construction as ranks and from that idea propose a new slice-based construction that is deterministic-in-the-limit and reaches optimality. They achieve this by defining a labeling function $\lambda^k : S_i \to \{\top, \bot\}$ where S_i is the set of nodes on level i of the run graph and parameter k is the guessed level after which all runs containing original accepting states must terminate. The authors call this a *retrospective* labeling because once the level k has been guessed, there are no further guesses made about the future, hence the determinism-in-the-limit. The tuple construction, albeit having been developed independently can also be seen as following a retrospective principle, which would probably be true of any construction that is deterministic-in-the-limit. Concretely however, no real similarities exist between that algorithm and the tuple construction, as the main outcome of Fogarty et al.'s findings is to use the λ^k labeling to define a new ranking function, allowing them to take advantage of known results (mainly regarding complexity) of rank-based algorithms. Whether this approach is competitive against our construction is subject of future research and would require implementing the algorithm in an efficient way.

5.2 Performance

Regardless of optimality, a desirable property of a complementation algorithm is of course performance on practical cases, which generally avoid the worst-case scenario. For that we use the same benchmarking protocol as in [13], i.e. we compare the number of states produced by our construction against the three major types of complementation algorithms: Slice-based [5], Rank-based [10], and Safra-Piterman [8] on the set of randomly generated automata used in the [13]. This collection holds a total of 11'000 automata having 15 states, a transition density ranging from 1.0 to 3.0, and a density of accepting states ranging from 0.1 to 1.0.

[7] Here the non-exponential part disappears from the O notation. We also simplify $\ln(\frac{\phi+1}{\phi})$ into $\ln(\phi)$.

For each algorithm we compare the number of states produced by the implementation found in the *GOAL* tool[8] [14] against our own (Java) implementation of the algorithm. We shall note that this implementation has been made available in the *GOAL* distribution by Daniel Weibel under the name "Fribourg construction" [16].

The exact tuple algorithm we choose for this benchmark is the one presented as A_2 in Section 4.1.1, with an extra optimization which aims at discarding all constructed states such that their state description does not contain a 0-colored component. We give no equivalence proof here, but it is easy to see that for a run of A_2 to permanently avoid accepting states, it has to contain an uninterrupted sequence of 0-colored components.

In the following tables we show the win ratio of our construction, as defined by:

$$\text{win-ratio} := \frac{\text{\# instances where tuple outputs fewer states}}{\text{\# non-timed-out instance}}$$

where the time-out has been set to 60 seconds[9]. The table cells where the tuple construction performs as well or better than the opponent are shaded.

5.2.1 Slice-based construction

The slice-based construction we use for comparison is the one described in [5], using these 3 optimizations proposed by *GOAL*[10]: turnwise cut-point construction, reduce out-degree, and merge adjacent 0-sets or *-sets.

The results are shown in Table 1. The ratio of instances where one of the algorithms times out does not exceed 3%. These results clearly show that the tuple construction performs better in nearly all cases. The bottom line is interesting, and can be explained by the fact that when the input automaton is universal, only the upper part of the tuple construction is left, making it equivalent to the slice-based construction in this case.

5.2.2 Rank-based construction

The rank-based construction against which we check our algorithm is the optimal version described in [10]. The following optimizations proposed by *GOAL* are used: tight-rank construction, turn-wise cut-point construction, and reduce out-degree.

Due to the very large number of states produced by that construction, we can only show partial results, and thus placed a dash in the table when the rank-based construction is unable to produce a significant number of complements within 60 seconds. The results are shown in Table 2.

Again, the tuple construction outperforms the rank-based in most cases. But it is interesting to see that there are areas where the latter produces smaller automata, namely when the transition density is low and the accepting density is high. That can probably be explained by the fact that with many accepting states in the input automaton, the tuple construction has to go through more steps to create an accepting state because it takes longer to eliminate all bad paths.

As with the slice-based, when the input is universal the two constructions produce the same number of states (which explains the line of 0's). This can be explained by the fact that a tuple is really a pre-order over the set of states, which can also be seen as a ranking. So the rank-based and the slice-based constructions degenerate to the same entity in that particular case.

5.2.3 Safra-Piterman construction

The third algorithm to be compared is the Safra-Piterman construction as described in [8]. It was described as giving the smallest complements in [13]. The following optimizations were used in our benchmark: use of Schewe's history trees instead of compact Safra trees, and reduction of transitions in the conversion from NPW to NBW based on the idea in the slice-based construction.

Again, we do not enable any extra pre- or post-processing that we do not also apply to the tuple construction. The number of instance where any of the algorithms timed out is within 3%. The results printed in Table 3 show that, although Safra-Piterman beats the tuple algorithm in most cases, there are some cases where the latter outperforms the former. This hints about the areas the construction can further be improved.

6 Conclusion

We present in this paper a complementation procedure for Büchi-automata that we claim is simple. By applying a construction based on tuples of subsets (similar to slices), we create a deterministic automaton that has the property that exactly those ω-words that are not accepted by the original automaton for which the run of the constructed automaton contains only finitely many state with components that consist of accepting states and are *continued*. Similar to the complementation of deterministic Büchi-automata, we create the complement automaton by taking the constructed deterministic automaton (the "upper" automaton) and a copy of it that we decorate with a coloring of state components (the "lower" automaton). An accepting continued component appears in a run of the lower automaton if and only if the run is non-accepting. Therefore by forcing the complement automaton to jump nondeterministically from the upper to the lower automaton, we guarantee that it only accepts ω-words that would create a run in the upper automaton in which states with continued accepting components recur finitely often (these are exactly the ω-words not accepted by the original automaton).

By applying two simple improvement to the coloring of the lower automaton, we can show that the resulting complement automaton can in the worst case show a growth in $O((0.7645n)^n)$, the known lower bound for the complement automaton. So the improved construction of the complement automaton is worst-case optimal (sometimes also called tight). In the existing literature, the factor 0.7645 in the worst-case bound was always estimated numerically. In this paper we can give an exact expression for it: $\frac{1}{\ln(\phi)\cdot e}$, where ϕ is the golden ratio and e is Euler's number.

An experimental evaluation shows that the worst-case optimal construction performs worse when applied to benchmark automata than the construction in which only one improvement of the state component coloring is applied (a construction that is not worst-case optimal). So we finally compare our complementation procedure that performs best on the benchmark automata with other known complementation algorithms for Büchi automata.

[8] http://goal.im.ntu.edu.tw

[9] each job was allocated one 2.57GHz processor and 2 GB memory.

[10] These heuristics allow reducing the size of the complement automata without having too big an effect on the execution time. Other optimizations are available, like maximizing the acceptance set of the input automaton, but these have not been switched on for fairness purposes, as we do not apply this kind of preprocessing on the tuple construction either.

	Transition density										
Acc. density	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0
0.1	94	100	98	99	99	100	100	100	99	98	99
0.2	95	99	97	100	100	100	100	100	100	100	100
0.3	97	97	99	100	100	100	100	100	100	100	100
0.4	89	96	100	100	100	100	100	100	100	99	100
0.5	93	96	100	100	99	100	100	100	100	100	100
0.6	87	100	99	99	100	100	100	100	100	100	100
0.7	76	88	98	95	100	100	100	100	100	100	100
0.8	69	81	91	98	100	100	100	100	100	100	100
0.9	55	63	84	99	100	100	100	100	100	100	100
1.0	0	0	0	0	0	0	0	0	0	0	0

Table 1. Win ratio in % of the tuple construction against the slice-based construction

	Transition density										
Acc. density	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0
0.1	-	-	-	-	-	-	-	-	-	-	-
0.2	-	-	-	-	-	-	-	-	-	-	-
0.3	-	-	-	-	-	-	-	-	-	-	-
0.4	-	-	-	-	-	-	-	-	-	-	-
0.5	91	89	-	-	-	-	-	-	-	-	-
0.6	71	73	86	84	99	97	100	100	100	100	100
0.7	40	28	22	40	61	79	94	98	99	99	100
0.8	39	19	6	23	42	50	87	90	97	100	100
0.9	22	13	4	6	26	42	62	72	88	88	100
1.0	0	0	0	0	0	0	0	0	0	0	0

Table 2. Win ratio in % of the tuple construction against the rank-based construction

	Transition density										
Acc. density	1.0	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0
0.1	55	37	17	12	15	17	6	14	11	15	16
0.2	51	27	21	9	9	8	6	4	10	7	10
0.3	37	14	12	5	4	4	2	2	5	4	8
0.4	40	9	3	0	1	1	4	6	5	6	8
0.5	41	9	3	0	1	4	0	7	3	4	15
0.6	33	10	3	1	1	1	2	5	7	9	11
0.7	30	7	3	4	4	9	11	10	19	25	32
0.8	40	12	6	6	14	15	18	21	38	41	43
0.9	36	16	8	2	12	19	32	46	54	58	77
1.0	99	94	82	61	32	23	17	5	9	2	3

Table 3. Win ratio in % of the tuple construction against the Safra-Piterman construction

References

[1] J. R. Büchi. 1962. On a Decision Method in Restricted Second Order Arithmetic. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science 1960*, E. Nagel et al. (Eds.). Stanford University Press, 1–11.

[2] Dana Fisman and Yoad Lustig. 2015. A Modular Approach for Büchi Determinization. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*. 368–382. https://doi.org/10.4230/LIPIcs.CONCUR.2015.368

[3] Seth Fogarty, Orna Kupferman, Thomas Wilke, and Moshe Y. Vardi. 2013. Unifying Büchi Complementation Constructions. *Logical Methods in Computer Science* 9, 1 (2013).

[4] O. A. Gross. 1962. Preferential Arrangements. *The American Mathematical Monthly* 69, 1 (1962), 4–8.

[5] Detlef Kähler and Thomas Wilke. 2008. Complementation, Disambiguation, and Determinization of Büchi Automata Unified. In *ICALP (1) (Lecture Notes in Computer Science)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.), Vol. 5125. Springer, 724–735.

[6] Orna Kupferman and Moshe Y. Vardi. 2001. Weak alternating automata are not that weak. *ACM Transactions on Computational Logic* 2 (July 2001), 408–429. Issue 3. https://doi.org/10.1145/377978.377993

[7] Frank Nießner, Ulrich Nitsche, and Peter Ochsenschläger. 1998. Deterministic ω-Regular Liveness Properties. In *Proceedings of the 3rd International Conference on Developments in Language Theory (DLT'97)*, Symeon Bozapalidis (Ed.). Thessaloniki, Greece, 237–247.

[8] Nir Piterman. 2006. From Nondeterministic Büchi and Streett Automata to Deterministic Parity Automata. In *Proceedings of the 21st Annual IEEE Symposium on Logic in Computer Science*. Seattle, WA, 255–264. https://doi.org/10.1109/LICS.2006.28

[9] Shmuel Safra. 1988. On the Complexity of ω-Automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*. White Plains.

[10] Sven Schewe. 2009. Büchi Complementation Made Tight. In *STACS (LIPIcs)*, Susanne Albers and Jean-Yves Marion (Eds.), Vol. 3. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 661–672.

[11] J. Stirling. 1730. *Methodus differentialis, sive tractatus de summation et interpolation serierum infinitarium*.

[12] Wolfgang Thomas. 1990. Automata on Infinite Objects. In *Formal Models and Semantics (Handbook of Theoretical Computer Science)*, Jan van Leeuwen (Ed.), Vol. B. Elsevier, 133–191.

[13] Ming-Hsien Tsai, Seth Fogarty, Moshe Y. Vardi, and Yih-Kuen Tsay. 2010. State of Büchi Complementation. In *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers (Lecture Notes in Computer Science)*, Michael Domaratzki and Kai Salomaa (Eds.), Vol. 6482. Springer, 261–271. https://doi.org/10.1007/978-3-642-18098-9_28

[14] Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Kang-Nien Wu, and Wen-Chin Chan. 2007. GOAL: A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In *Tools and Algorithms for the Construction and Analysis of Systems*, Orna Grumberg and Michael Huth (Eds.). Lecture Notes in Computer Science, Vol. 4424. Springer Berlin Heidelberg, 466–471. https://doi.org/10.1007/978-3-540-71209-1_35

[15] Ulrich Ultes-Nitsche. 2007. A power-set construction for reducing Büchi automata to non-determinism degree two. *Information Processing Letters (IPL)* 101, 3 (February 2007), 107–111.

[16] Daniel Weibel. 2015. Empirical Performance Investigation of a Büchi Complementation Construction. Master thesis, University of Fribourg, Switzerland. (2015).

[17] Qiqi Yan. 2008. Lower Bounds for Complementation of omega-Automata Via the Full Automata Technique. *Journal of Logical Methods in Computer Science* 4, 1 (2008), 20 pages.