

COMPUTING THE WIDTH OF NON-DETERMINISTIC AUTOMATA

DENIS KUPERBERG AND ANIRBAN MAJUMDAR

CNRS, LIP, ENS Lyon
e-mail address: denis.kuperberg@ens-lyon.fr

LSV, CNRS & ENS Paris-Saclay, Univ. Paris-Saclay – Cachan (France)
Univ. Rennes, Inria, CNRS, IRISA – Rennes (France)
e-mail address: majumdar@lsv.fr

ABSTRACT. We introduce a measure called width, quantifying the amount of nondeterminism in automata. Width generalises the notion of good-for-games (GFG) automata, that correspond to NFAs of width 1, and where an accepting run can be built on-the-fly on any accepted input. We describe an incremental determinisation construction on NFAs, which can be more efficient than the full powerset determinisation, depending on the width of the input NFA. This construction can be generalised to infinite words, and is particularly well-suited to coBüchi automata. For coBüchi automata, this procedure can be used to compute either a deterministic automaton or a GFG one, and it is algorithmically more efficient in the last case. We show this fact by proving that checking whether a coBüchi automaton is determinisable by pruning is NP-complete. On finite or infinite words, we show that computing the width of an automaton is EXPTIME-complete. This implies EXPTIME-completeness for multi-pebble simulation games on NFAs.

1. INTRODUCTION

Determinisation of non-deterministic automata (NFAs) is one of the cornerstone problems of automata theory. Determinisation algorithms occupy a central place in the theoretical study of regular languages of finite or infinite words, inducing for instance many of the robustness properties of these classes. Moreover, determinisation algorithms are not only used to prove theoretical properties related with decidability and complexity, but are also used when we want to put these theories to practical use, with many applications for instance in verification and synthesis. Consequently, there is a very active field of research aiming at optimizing or approximating determinisation, or circumventing it in contexts like inclusion of NFA or Church Synthesis. Indeed, determinisation is a costly operation, as the state space blow-up is in $O(2^n)$ on finite words, $O(3^n)$ for coBüchi automata [25], and $2^{O(n \log(n))}$ for Büchi automata [29], where there is also an increased complexity of the acceptance condition, going from Büchi to Rabin.

If \mathcal{A} and \mathcal{B} are NFAs, the classical way of checking the inclusion $L(\mathcal{A}) \subseteq L(\mathcal{B})$ is to determinise \mathcal{B} , complement it, and test emptiness of $L(\mathcal{A}) \cap \overline{L(\mathcal{B})}$. To circumvent a full

Work supported by the Grant Palse Impulsion.

determinisation, the recent algorithm from [6] proved to be very efficient, as it is likely to explore only a part of the powerset construction. Other approaches use simulation games to approximate inclusion at a cheaper cost, see for instance [14].

Another way of avoiding a full determinisation construction consists in replacing determinism by a weaker constraint that suffices in some particular contexts. In this spirit, Good-for-Games (GFG for short) automata were introduced in [16], as a way to solve the Church synthesis problem. This problem [8] asks, given a specification L , typically given by an LTL formula over an alphabet of inputs and outputs, whether there is a reactive system (transducer) whose behaviour is included in L . The classical solution [27] computes a deterministic automaton for L , and solves a game defined on this automaton. It turns out that replacing determinism by the weaker constraint of being GFG is sufficient in this context. Intuitively, a GFG automaton is a non-deterministic automaton where it is possible to build an accepting run in an online way, without any knowledge of the future, provided the input word is in the language of the automaton. In [16], it is shown that GFG automata allow an incremental algorithm for the Church synthesis problem: we can build increasingly large games, with the possibility that the algorithm stops before the full determinisation is needed. One of the aims of this paper is to generalise this idea to determinisation of NFA, for use in any context and not only Church synthesis. We give an incremental determinisation construction, where the emphasis is on space-saving, and that allows in some cases to avoid the full powerset construction.

The notion of width introduced in [19] (of which this paper is an extended version) generalises the GFG model, by allowing more than one run to be built in an online way. Intuitively, width quantifies how many states we have to keep track of simultaneously in order to build an accepting run in an online way. The maximal width of an automaton is its number of states. The width of an automaton also corresponds to the number of steps performed by our incremental determinisation construction before stopping. In the worst case where the width is equal to the number of states of the automaton, we end up performing the full powerset construction (or its generalisations for infinite words). We study here the complexity of directly computing the width of a nondeterministic automaton, and we show that it is EXPTIME-complete, even in the restricted case of universal safety automata. This constitutes a new contribution compared to the conference version of this paper [19], where only PSPACE-hardness was shown for the width problem.

We obtain this result via a reduction from a combinatorial game on boolean formulas from [30]. In the process, we also show that multi-pebble simulation games on NFAs are EXPTIME-complete, even when testing simulation of a trivial automaton by an NFA of size n , using a fixed number of $n/2$ pebbles. This generalizes a previous result from [9], where EXPTIME-completeness is shown for multi-pebble simulations on Büchi automata, with a number of pebbles fixed to \sqrt{n} .

The properties of GFG automata and their links with other models and algorithms (tree automata, Markov Decision Processes, efficient algorithms for parity games) are nowadays actively investigated [4, 18, 20, 5, 2, 28, 17, 12]. Colcombet introduced a generalisation of the concept of GFG called history-determinism [10], replacing determinism in the framework of automata with counters. It was conjectured by Colcombet [11] that GFG automata were essentially deterministic automata with additional useless transitions. It was shown in [20] that on the contrary there is in general an exponential state space blowup to translate GFG automata to deterministic ones. GFG automata retain several good properties of

determinism, in particular they can be composed with trees and games, and easily checked for inclusion.

We give here the first algorithms allowing to build GFG automata from arbitrary non-deterministic automata on infinite words, allowing to potentially save exponential space compared to deterministic automata. Our incremental constructions look for small GFG automata, and aim at avoiding the worst-case complexities of determinisation constructions. Moreover, in the case of coBüchi automata, we show that the procedure is more efficient than its analog looking for a deterministic automaton, since checking the GFG property is polynomial [20], while we show here that the corresponding step for determinisation, that is checking whether a coBüchi automaton is Determinisable By Pruning (DBP) is NP-complete. Combined with the good properties of GFG coBüchi automata related to succinctness even for LTL-definable languages [17] and polynomial time minimization [28], this makes the class of coBüchi automata especially well-suited for this approach.

As a measure of non-determinism, width can be compared with ambiguity, where the idea is to limit the number of possible runs of the automaton. In this context unambiguous automata play a role analogous to GFG automata for width. Unambiguous automata are studied in [23], degrees of ambiguity are investigated in [31, 21, 22]. We give examples of automata with various width and ambiguity, showing that these two measures are essentially orthogonal.

After defining automata and games in Section 2, we describe the width approach on finite words and the incremental determinisation construction for NFAs in Section 3. We compare width and ambiguity in Section 4, and show a link between width and multi-pebble simulation relations in Section 5. We show in Section 6 that computing the width of a NFA, as well as testing whether a multi-pebble simulation holds, is EXPTIME-complete. We then move to infinite words, and start by focusing on the coBüchi acceptance condition in Section 7. We show that the breakpoint construction [25] can be adapted to yield an incremental breakpoint construction, that can be used to build either a deterministic or a GFG coBüchi automaton from a nondeterministic one. We compare the two approaches, and exhibit several advantages of GFG automata in this special case. We finally describe the general case of Büchi automata in Section 8, where we give an incremental version of the Safra construction [29], and point to open problems related to the algorithmic complexity of this approach.

2. DEFINITIONS

We will use Σ to denote a finite alphabet. The empty word is denoted ε . If $i \leq j$, the set $\{i, i+1, i+2, \dots, j\}$ is denoted $[i, j]$. If X is a set and $k \in \mathbb{N}$, we note $X^{\leq k}$ the set of subsets of X of size at most k . The complement of a set X is denoted \overline{X} . If $u \in \Sigma^*$ is a word and $L \subseteq \Sigma^*$ is a language, the left quotient of L by u is $u^{-1}L := \{v \in \Sigma^* \mid uv \in L\}$.

2.1. Automata. A non-deterministic automaton \mathcal{A} is a tuple $(Q, \Sigma, q_0, \Delta, F)$ where Q is the set of states, Σ is a finite alphabet, $q_0 \in Q$ is the initial state, $\Delta : Q \times \Sigma \rightarrow 2^Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states.

The transition function is naturally generalised to 2^Q by setting for any $(X, a) \in 2^Q \times \Sigma$, $\Delta(X, a)$ the set of a -successors of X , i.e. $\Delta(X, a) = \{q \in Q \mid \exists p \in X, q \in \Delta(p, a)\}$.

We will sometimes identify Δ with its graph, and write $(p, a, q) \in \Delta$ instead of $q \in \Delta(p, a)$.

If for all $(p, a) \in Q \times \Sigma$ there is a unique $q \in Q$ such that $(p, a, q) \in \Delta$, we say that \mathcal{A} is *deterministic*.

If $u = a_1 \dots a_n$ is a finite word of Σ^* , a run of \mathcal{A} on u is a sequence $q_0 q_1 \dots q_n$ such that for all $i \in [1, n]$, we have $q_i \in \Delta(q_{i-1}, a_i)$. The run is said to be *accepting* if $q_n \in F$.

If $u = a_1 a_2 \dots$ is an infinite word of Σ^ω , a run of \mathcal{A} on u is a sequence $q_0 q_1 q_2 \dots$ such that for all $i > 0$, we have $q_i \in \Delta(q_{i-1}, a_i)$. A run is said to be *Büchi accepting* if it contains infinitely many accepting states, and *coBüchi accepting* if it contains finitely many non-accepting states. Automata on infinite words will be called Büchi and coBüchi automata, to specify their acceptance condition.

We will note NFA (resp. DFA) for a non-deterministic (resp. deterministic) automaton on finite words, NBA (resp. DBA) for a non-deterministic (resp. deterministic) Büchi automaton, and NCA (resp. DCA) for a non-deterministic (resp. deterministic) coBüchi automaton.

We also mention more general acceptance conditions on infinite words:

- *Parity condition*: each state q has a rank $\text{rk}(q) \in \mathbb{N}$, and an infinite run is accepting if the highest rank appearing infinitely often is even.
- *Rabin condition*: a set $\{(G_1, B_1), \dots, (G_k, B_k)\}$ of pairs with $G_i, B_i \subseteq Q$ is given, a run is accepting if there exists $i \in [1, k]$ such that the run contains infinitely many states from G_i and finitely many states from B_i .
- *Streett condition*: dual of the Rabin condition.
- *Muller condition*: a set $\{F_1, \dots, F_k\}$ of subsets of Q is given, a run is accepting if there is $i \in [1, k]$ such that the set of states appearing infinitely often is exactly F_i .

The language of an automaton \mathcal{A} , noted $L(\mathcal{A})$, is the set of words on which the automaton \mathcal{A} has an accepting run. Two automata are said *equivalent* if they recognise the same language. An automaton \mathcal{A} is said *universal* if it accepts all words.

An automaton \mathcal{A} is *determinisable by pruning* (DBP) if an equivalent deterministic automaton can be obtained from \mathcal{A} by removing some transitions.

An automaton \mathcal{A} is *Good-For-Games* (GFG) if there exists a function $\sigma: \Sigma^* \rightarrow Q$ (called *GFG strategy*) that resolves the non-determinism of \mathcal{A} depending only on the prefix of the input word read so far: over every word $u = a_1 a_2 a_3 \dots$ (finite or infinite depending on the type of automaton considered), the sequence of states $\sigma(\varepsilon)\sigma(a_1)\sigma(a_1 a_2)\sigma(a_1 a_2 a_3) \dots$ is a run of \mathcal{A} on u , and it is accepting whenever $u \in L(\mathcal{A})$. For instance every DBP automaton is GFG. See [4] for more introductory material and examples on GFG automata.

2.2. Games. A *game* $\mathcal{G} = (V_0, V_1, v_I, E, W)$ of infinite duration between two players 0 and 1 consists of: a finite set of *positions* V being a disjoint union of V_0 and V_1 ; an *initial position* $v_I \in V$; a set of *edges* $E \subseteq V \times V$; and a *winning condition* $W \subseteq V^\omega$.

A *play* is an infinite sequence of positions $v_0 v_1 v_2 \dots \in V^\omega$ such that $v_0 = v_I$ and for all $n \in \mathbb{N}$, $(v_n, v_{n+1}) \in E$. A play $\pi \in V^\omega$ is *winning* for Player 0 if it belongs to W . Otherwise π is *winning* for Player 1.

A *strategy* for Player 0 (resp. 1) is a function $\sigma_0: V^* \times V_0 \rightarrow V$ (resp. $\sigma_1: V^* \times V_1 \rightarrow V$), describing which edge should be played given the history of the play $u \in V^*$ and the current position $v \in V$. A strategy σ_P has to obey the edge relation, i.e. there has to be an edge in

E from v to $\sigma_P(u, v)$. A play $\pi = v_0v_1v_2\dots$ is *consistent* with a strategy σ_P of a player P if for every n such that $v_n \in V_P$ we have $v_{n+1} = \sigma_P(v_0\dots v_{n-1}, v_n)$.

A strategy for Player 0 (resp. Player 1) is *positional* if it does not use the history of the play, i.e. it can be seen as a function $V_0 \rightarrow V$ (resp. $V_1 \rightarrow V$).

We say that a strategy σ_P of a player P is *winning* if every play consistent with σ_P is winning for P . In this case, we say that P *wins* the game \mathcal{G} .

A game is *positionally determined* if exactly one of the players has a positional winning strategy in the game.

3. FINITE WORDS

3.1. Width of an NFA. We want to define the *width* of an NFA as the minimum number of simultaneous states that need to be tracked in order to be able to deterministically build an accepting run in an online way. Let $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ be an NFA, and $n = |Q|$ be the size of \mathcal{A} . In order to define this notion of width formally, we introduce a family of games $\mathcal{G}_w(\mathcal{A}, k)$, parameterized by an integer $k \in [1, n]$.

The game $\mathcal{G}_w(\mathcal{A}, k)$ is played on $Q^{\leq k}$, starts in $X_0 = \{q_0\}$, and the round i of the game from a position $X_i \in Q^{\leq k}$ is defined as follows:

- Player 1 chooses a letter $a_{i+1} \in \Sigma$.
- Player 0 moves to a subset $X_{i+1} \subseteq \Delta(X_i, a_{i+1})$ of size at most k .

A play is winning for Player 0 if for all $r \in \mathbb{N}$, whenever $a_1a_2\dots a_r \in L(\mathcal{A})$, X_r contains an accepting state.

Definition 3.1. The width of an NFA \mathcal{A} , denoted $\text{width}(\mathcal{A})$, is the least k such that Player 0 wins $\mathcal{G}_w(\mathcal{A}, k)$.

Intuitively, the width measures the “amount of non-determinism” in an automaton: it counts the number of simultaneous states we have to keep track of, in order to be sure to find an accepting run in an online way.

Fact 3.2. An NFA \mathcal{A} is GFG if and only if $\text{width}(\mathcal{A}) = 1$.

3.2. Partial powerset construction. We give here a generalisation of the powerset construction, following the intuition of the width measure.

We define the k -subset construction of \mathcal{A} to be the subset construction where the size of each set is bounded by k . Formally, it is the NFA $\mathcal{A}_k = (Q^{\leq k}, \Sigma, \{q_0\}, \Delta', F')$ where:

- $\Delta'(X, a) := \begin{cases} \{\Delta(X, a)\} & \text{if } |\Delta(X, a)| \leq k \\ \{X' \mid X' \subseteq \Delta(X, a), |X'| = k\} & \text{otherwise} \end{cases}$
- $F' := \{X \in Q^{\leq k} \mid X \cap F \neq \emptyset\}$

Remark 3.3. Notice that \mathcal{A}_1 is isomorphic to the automaton \mathcal{A} .

Lemma 3.4. *The automaton \mathcal{A}_k has less than $\frac{n^k}{(k-1)!} + 1$ states.*

Proof. The number of states of \mathcal{A}_k is (at most) $|Q^{\leq k}| = \sum_{i=0}^k \binom{n}{i}$. Using the fact that $\binom{n}{i} \leq \frac{n^i}{i!}$, we can bound the number of states of \mathcal{A}_k by $\sum_{i=0}^k \frac{n^i}{i!} \leq \sum_{i=0}^k \frac{n^k}{k!} \leq 1 + \sum_{i=1}^k \frac{n^k}{k!} = \frac{n^k}{(k-1)!} + 1$. \square

The following lemma shows the link between width and the k -powerset construction.

Lemma 3.5. *One has $\text{width}(\mathcal{A}) \leq k$ if and only if \mathcal{A}_k is GFG.*

Proof. Winning strategies in $\mathcal{G}_w(\mathcal{A}, k)$ are in bijection with GFG strategies for \mathcal{A}_k . \square

3.3. GFG automata on finite words. We recall here results on GFG automata on finite words.

We start with a lemma characterizing GFG strategies. Let $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ be an NFA recognising a language L , and $\sigma : \Sigma^* \rightarrow Q$ be a potential GFG strategy. For any $q \in Q$, we denote $L(q)$ the language accepted from q in \mathcal{A} , i.e. $L(q)$ is the language of \mathcal{A} with q as initial state.

Lemma 3.6. *σ is a GFG strategy if and only if for all $u \in \Sigma^*$, $L(\sigma(u)) = u^{-1}L$.*

Proof. Assume σ is a GFG strategy, and let $u \in \Sigma^*$. Let $q = \sigma(u)$. It is clear that $L(q) \subseteq u^{-1}L$, as any run accepting v from q is a witness that $uv \in L$ (together with the run on u reaching q from q_0). We therefore have to show that for all $v \in u^{-1}L$, we have $v \in L(q)$. For this, recall that σ is a GFG strategy, so $\sigma(uv) \in F$. Since $\sigma(u) = q$, there is an accepting run starting in q and labelled by v , showing $v \in L(q)$.

Conversely, assume that for any $u \in \Sigma^*$, $L(\sigma(u)) = u^{-1}L$. In particular, it means, that for any $u \in L$ we have $\varepsilon \in L(\sigma(u))$, so $\sigma(u)$ is an accepting state. This implies that σ is indeed a GFG strategy. \square

We now go to the main result of this section. This result on DBP automata has first been proved in [1], and then a more general version allowing lookahead was proved using a game-based approach in [24]. The link between GFG and DBP automata on finite words was first mentioned in [11].

Theorem 3.7. [1, 11, 24] *An NFA \mathcal{A} is GFG if and only if it is DBP. Moreover, there is a quadratic algorithm that determines whether an NFA is GFG, and in the positive cases builds an equivalent DFA by removing transitions.*

3.4. Incremental determinisation procedure. We can now describe an incremental determinisation procedure, aiming at saving resources in the search of a deterministic automaton. In the process, we also compute the width of the input NFA.

The algorithm goes as follows:

The usual determinisation procedure uses the full powerset construction, i.e. assumes that we are in the case of maximal width. Once a deterministic automaton has been obtained, be it by full determinization or via our incremental approach, it can be minimized easily.

Our method here is to approach this powerset construction “from below”, and incrementally increase the width until we find the good one. In some cases, this allows to compute directly a smaller automaton, and avoids using the full powerset construction of exponential state complexity as an intermediary step.

Algorithm 1: Incremental NFA determinisation

Input: NFA \mathcal{A} **Output:** $\text{width}(\mathcal{A})$ and DFA \mathcal{D} equivalent to \mathcal{A} $k := 1;$ **while** \mathcal{A}_k *is not* GFG **do** $k := k + 1;$ Construct \mathcal{A}_k ; Compute an equivalent DFA \mathcal{D} from \mathcal{A}_k by removing transitions; Return k, \mathcal{D} ;

For an NFA with n states and width k , the complexity of this algorithm is in $O\left(\frac{n^{2k}}{(k-1)!^2}\right)$, by Lemma 3.4 and Theorem 3.7.

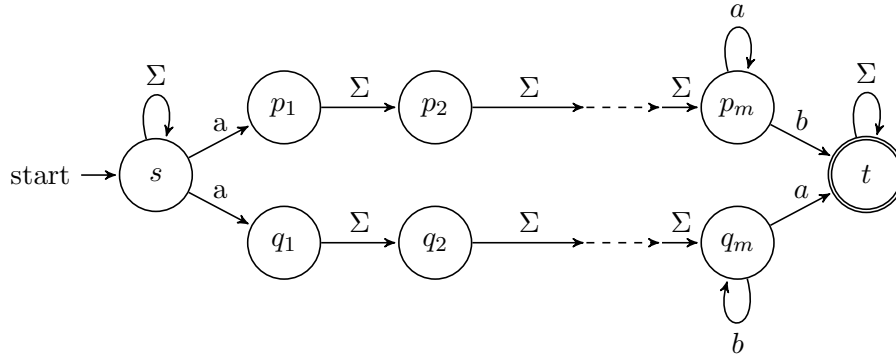


Figure 1: Example: 2-subset construction is enough

Example 3.8. We take an example in Figure 1. Here the language recognised by this automaton is $L(\mathcal{A}) = \Sigma^* a \Sigma^{\geq m}$, and it has width 2. Indeed, the automaton \mathcal{A}_2 is DBP, and can be pruned to keep only states $\{s\}, \{p_1, q_1\}, \dots, \{p_m, q_m\}, \{t\}$ (so getting rid of states such as $\{s, p_1\}$) while still recognizing $L(\mathcal{A})$. Therefore, our determinisation procedure uses time $O(n^4)$ and builds an intermediary DFA \mathcal{A}_2 of size $O(n^2)$, while a classical determinisation via powerset construction would build an exponential-size DFA. The pruning process of the DBP automaton \mathcal{A}_2 yields here the minimal DFA of size $m + 2 = O(n)$.

But in some other cases, the powerset construction is actually more efficient than the k -powerset construction, in terms of number of reachable states. For instance consider an example where the alphabet is $\Sigma = \{a_1, a_2, \dots, a_n\}$ and the automaton has $n + 2$ states: one initial state, one final state and n other transition states, as shown in Figure 2. The transition relation is defined as in the picture. On this example, the automaton obtained from subset construction has only 3 states whereas for any k , the automaton obtained by k -subset construction will have $\binom{n}{k} + 2$ states. This example illustrates that sometimes the powerset construction can actually be more efficient than the k -powerset construction, and the incremental k -powerset construction is not necessarily increasing in terms of number of states as k grows. It would therefore be interesting to be able to either run the two methods in parallel, or guess which one is more efficient based on the shape of the input NFA.

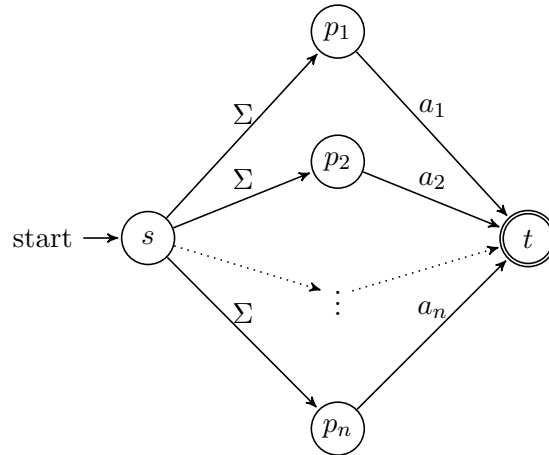


Figure 2: Example: Subset construction can be efficient

4. WIDTH VERSUS AMBIGUITY

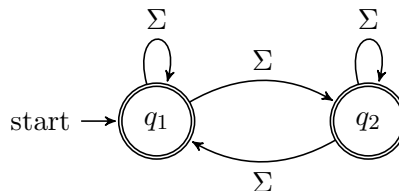
In this section, we recall a useful notion of automata, namely *ambiguity* from [21] and investigate its relation with the notion of *width* in the form of examples.

Definition 4.1. Given an NFA \mathcal{A} and a word w , the ambiguity of w is the number of different accepting paths for w in \mathcal{A} .

Note that a word is accepted by \mathcal{A} if and only if the ambiguity of the word is non-zero. \mathcal{A} is called *unambiguous* if ambiguity of any word is either zero or one. \mathcal{A} is called *finitely* (resp. *polynomially*, *exponentially*) *ambiguous* if there exists a constant (resp. polynomial, exponential) function f such that the ambiguity of any word of length n is bounded by $f(n)$.

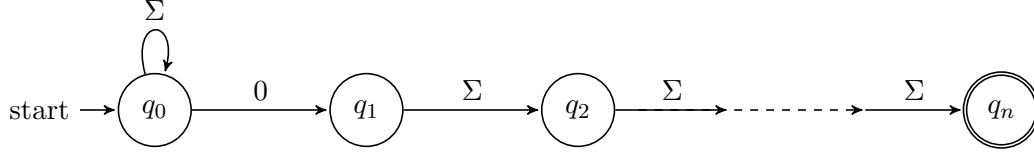
For example, every DFA is unambiguous since every word accepted by a DFA has a unique accepting run. Some more illustrated examples are given in this section, showing that width and ambiguity can vary independently from each other in NFAs.

4.1. Width 1, Exponentially ambiguous. Consider the following NFA accepting all words in Σ^* .



The above automaton is exponentially ambiguous but not polynomially ambiguous. Indeed each word of length n has 2^n accepting runs. However it has width 1, since it suffices to stay in q_1 to produce an accepting run.

4.2. Width n , Unambiguous. Consider the following NFA \mathcal{A}_n , recognizing the language $\Sigma^*0\Sigma^{n-1}$.

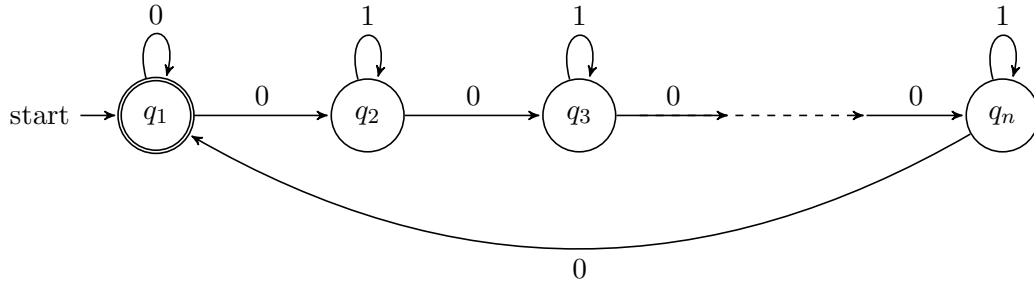


Every word which is in the language of this automaton is accepted by a unique run of \mathcal{A}_n . Therefore, it is an unambiguous automaton.

But one can show that the minimal DFA for \mathcal{A}_n has exactly 2^n states. By Theorem 3.7, this implies that this automaton has width n . Indeed, if the width was $k < n$, we could build a deterministic automaton with strictly less than 2^n states.

More precisely, this automaton has $n + 1$ states and width n , so this is an example of an NFA \mathcal{A}_n of width $|\mathcal{A}_n| - 1$.

4.3. Width n , Exponentially ambiguous. Consider the following NFA \mathcal{A}_n , recognizing to the language $L_n = (0 + (01^*)^{n-1}0)^*$.



It is shown in [21] that \mathcal{A}_n is exponentially ambiguous but not polynomially ambiguous, and that any DFA (actually any polynomially ambiguous NFA) recognising L_n must have $2^n - 1$ states. Therefore, \mathcal{A}_n has width n by Theorem 3.7, as in the previous example.

5. RELATION WITH MULTIPLEBBLE SIMULATIONS

5.1. Multiplebble Simulation. Let $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma, q_{\mathcal{A}}^0, \Delta_{\mathcal{A}}, F_{\mathcal{A}})$ and $\mathcal{B} = (Q_{\mathcal{B}}, \Sigma, q_{\mathcal{B}}^0, \Delta_{\mathcal{B}}, F_{\mathcal{B}})$ be NFAs, and k be a positive integer. The k -simulation game $\mathcal{G}_k(\mathcal{A}, \mathcal{B})$ between Spoiler and Duplicator is defined as follows.

The game is played on arena $Q_{\mathcal{A}} \times (Q_{\mathcal{B}})^{\leq k}$. The initial position is $(q_{\mathcal{A}}^0, \{q_{\mathcal{B}}^0\})$.

A round from position (p, X) consists in the following moves:

- Spoiler plays a transition $(p, a, p') \in \Delta_{\mathcal{A}}$
- Duplicator chooses $X' \subseteq \Delta_{\mathcal{B}}(X, a)$, with $|X'| \leq k$.
- the game moves to position (p', X') .

A position (p, X) is winning for Spoiler if $p \in F_{\mathcal{A}}$ but $X \cap F_{\mathcal{B}} = \emptyset$. Duplicator wins any play avoiding positions that are winning for Spoiler.

Definition 5.1 [14]. The k -simulation relation $\mathcal{A} \sqsubseteq_k \mathcal{B}$ is said to hold if Duplicator wins $\mathcal{G}_k(\mathcal{A}, \mathcal{B})$.

We visualize positions of the game via *pebbles*: in a position (p, X) , Spoiler has a pebble in the state p of \mathcal{A} , while Duplicator has pebbles in each state of the set X . Notice that according to the definition of the game, Duplicator can duplicate pebbles while erasing some others, as long as it owns at most k pebbles at every step. In other words it is not required that each pebble follows a particular run of the automaton. The relations \sqsubseteq_k can be used to approximate inclusion. Indeed, we have for any NFAs \mathcal{A}, \mathcal{B} [14]:

$$\mathcal{A} \sqsubseteq_1 \mathcal{B} \Rightarrow \mathcal{A} \sqsubseteq_2 \mathcal{B} \Rightarrow \cdots \Rightarrow \mathcal{A} \sqsubseteq_{|\mathcal{B}|} \mathcal{B} \Leftrightarrow L(\mathcal{A}) \subseteq L(\mathcal{B}).$$

Moreover, for fixed k the \sqsubseteq_k relation can be computed in polynomial time:

Theorem 5.2. [14] *There is an algorithm with inputs $\mathcal{A}, \mathcal{B}, k$ deciding whether $\mathcal{A} \sqsubseteq_k \mathcal{B}$ with time complexity $n^{O(k)}$, where $n = |\mathcal{A}| + |\mathcal{B}|$.*

We show in Section 6 that this problem is EXPTIME-complete, so this algorithm is optimal in the sense that it cannot avoid the exponent in k .

5.2. Width versus k -simulation. Links between width and k -simulation relations are explicated by the following two lemmas.

The first one shows that knowing the width of an NFA allows to use multi-pebble simulation to test for real inclusion of languages.

Lemma 5.3. *Let \mathcal{A}, \mathcal{B} be NFAs and $k = \text{width}(\mathcal{B})$. Then $L(\mathcal{A}) \subseteq L(\mathcal{B})$ if and only if $\mathcal{A} \sqsubseteq_k \mathcal{B}$.*

Proof. The right-to-left implication is true regardless of the value of k . It is already stated in [14], and follows from the fact that if $\mathcal{A} \sqsubseteq_k \mathcal{B}$, then any accepting run of \mathcal{A} chosen by Spoiler can be answered with a set of runs from \mathcal{B} containing an accepting one.

We show the converse, and assume $L(\mathcal{A}) \subseteq L(\mathcal{B})$. Let σ be a winning strategy for Player 0 in $\mathcal{G}_w(\mathcal{B}, k)$, witnessing that $k = \text{width}(\mathcal{B})$. Then Duplicator can also play σ in $\mathcal{G}_k(\mathcal{A}, \mathcal{B})$, ignoring the position of the pebble of Spoiler in \mathcal{A} . Since any word reaching an accepting state of \mathcal{A} is in $L(\mathcal{A}) \subseteq L(\mathcal{B})$, the strategy σ guarantees that at least one pebble of Duplicator is in an accepting state, by definition of σ . So this strategy is winning in $\mathcal{G}_k(\mathcal{A}, \mathcal{B})$, witnessing $\mathcal{A} \sqsubseteq_k \mathcal{B}$. \square

The second lemma shows a link in the other direction: width can be computed from the relations \sqsubseteq_k .

Lemma 5.4. *Let \mathcal{A} be an NFA and \mathcal{A}_{det} be a DFA for $L(\mathcal{A})$. Then for any $k \geq 1$, we have $\text{width}(\mathcal{A}) \leq k$ if and only if $\mathcal{A}_{\text{det}} \sqsubseteq_k \mathcal{A}$.*

Proof. Moves of Spoiler and Duplicator in $\mathcal{G}_k(\mathcal{A}_{\text{det}}, \mathcal{A})$ are in bijection with those of Player 1 and Player 0 respectively in $\mathcal{G}_w(\mathcal{A}, k)$. Indeed, for Spoiler, choosing a transition in \mathcal{A}_{det} amounts to only choosing a letter, since the state of \mathcal{A}_{det} is updated deterministically. Moves of Duplicator and Player 0 are identical in both games. The winning conditions also match in both games, since in a given round, the current state of \mathcal{A}_{det} is accepting if and only if the word played so far is in $L(\mathcal{A}_{\text{det}}) = L(\mathcal{A})$. Therefore, Duplicator wins $\mathcal{G}_k(\mathcal{A}_{\text{det}}, \mathcal{A})$ if and only if Player 0 wins $\mathcal{G}_w(\mathcal{A}, k)$, using the same strategy. \square

Notice that this does not imply a polynomial reduction between the width problem and multi-pebble simulation one way or another, since the size of \mathcal{A}_{det} is in general exponential in the size of \mathcal{A} .

We recall that an automaton accepting all words is said *universal*.

Corollary 5.5. *Let \mathcal{A} be a universal NFA, and $k \geq 1$. We have $\text{width}(\mathcal{A}) \leq k$ if and only if $\mathcal{A}_{\text{triv}} \sqsubseteq_k \mathcal{A}$, where $\mathcal{A}_{\text{triv}}$ is the trivial one-state automaton accepting all words.*

This corollary means that computing the width k of a universal NFA is as hard as testing its multi-pebble simulations up to k against $\mathcal{A}_{\text{triv}}$.

We will make use of this connection in the following, to show that both the width problem and the multi-pebble simulation testing are EXPTIME-complete.

6. COMPLEXITY RESULTS ON THE WIDTH PROBLEM

In this section, we study the complexity of the *width problem*: given an NFA \mathcal{A} and an integer k , is $\text{width}(\mathcal{A}) \leq k$?

Being able to solve this problem efficiently would allow us to optimize the incremental determinisation algorithm, by aiming at the optimal k matching the width right away instead of trying different width candidates incrementally.

The main theorem of this section is the following:

Theorem 6.1. *The width problem is EXPTIME-complete.*

We start by showing the upper bound:

Lemma 6.2. *The width problem is in EXPTIME.*

Proof. To show the EXPTIME upper bound, it suffices to build the game $\mathcal{G}_w(\mathcal{A}, k)$ of exponential size. This is a safety game, so solving it is polynomial in the size of the game. This means this algorithm runs in exponential time. Also note that Algorithm 1 given in Section 3.4 computes the width of an NFA in EXPTIME. \square

The rest of the section is devoted to showing the EXPTIME-hardness of the width problem. We will actually show a stronger result: the width problem is EXPTIME-hard on universal safety automata, i.e. automata with all states accepting, and where all words are accepted. By Corollary 5.5, this implies that this EXPTIME-hardness result applies also to deciding whether a multi-pebble simulation holds. We will proceed by reduction from a combinatorial game on boolean formulas, shown EXPTIME-complete in [30] (where it is named the game G_1). We will call this combinatorial game \mathcal{G}_c , and we start by describing it in the following.

6.1. The combinatorial game \mathcal{G}_c . An instance of the game \mathcal{G}_c is a tuple $(\varphi, X_0, X_1, \alpha_{\text{init}})$, where X_0 and X_1 are disjoint sets of variables, and φ is a 4-CNF formula on variables $V = X_0 \cup X_1 \cup \{t\}$, where $t \notin X_0 \cup X_1$, and finally α_{init} is a valuation $V \rightarrow \{0, 1\}$.

This means that φ is of the form $C_1 \wedge C_2 \wedge \dots \wedge C_n$, where $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3} \vee l_{i,4}$, in which each $l_{i,j}$ is a *literal*, i.e. a variable $x \in V$ or its negation \bar{x} . We will call \bar{V} the set $\{\bar{v} \mid v \in V\}$, and $\text{Lit} = V \cup \bar{V}$ the set of literals. Similarly, we define $\text{Lit}_i = X_i \cup \bar{X}_i$ for $i \in \{0, 1\}$.

A position in \mathcal{G}_c is of the form (τ, α) , where $\tau \in \{0, 1\}$ identifies the player who owns the position, and α is a valuation $V \rightarrow \{0, 1\}$.

In such a position, the player τ owning the position can change the values of variables in X_τ , and additionally the variable t is set to τ . This yields a new valuation α' . If this

valuation makes the formula φ false, Player τ immediately loses, otherwise the game moves to position $(1 - \tau, \alpha')$.

The starting position of the game is $(1, \alpha_{init})$. We say that Player 1 wins the game if he can force a win, i.e. if he has a strategy σ such that all plays compatible with σ eventually end with Player 0 losing the game by making the formula φ false.

It is shown in [30] that determining whether Player 0 wins a given instance of the game \mathcal{G}_c is EXPTIME-complete.

6.2. Reduction to the width problem. We now want to show that \mathcal{G}_c can be encoded in the width problem for a universal safety automaton. Let $I_c = (\varphi, X_0, X_1, \alpha_{init})$ be an instance of \mathcal{G}_c . We want to build an instance \mathcal{A}, k of the width problem such that $\text{width}(\mathcal{A}) \leq k$ if and only if Player 0 wins \mathcal{G}_c on instance I_c . Moreover the instance \mathcal{A}, k must be computable in polynomial time from I_c , and we want \mathcal{A} to be a universal safety automaton. We will reuse the notations of the previous section for describing the instance I_c . In particular $V = X_0 \cup X_1 \cup \{t\}$, and $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_n$, where $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3} \vee l_{i,4}$. The width we will be aiming for is $k = |V|$, i.e. the number of variables in φ .

6.2.1. Intuitive account of the construction. Before giving a formal definition of \mathcal{A} , let us sketch the intuitions guiding this construction. We want the width game of \mathcal{A} to mimic the game \mathcal{G}_c . This means that a subset of states of \mathcal{A} of size k will correspond to a valuation of k variables. Truth value of variables from X_0 will be chosen by Player 0 through nondeterminism in \mathcal{A} , while truth value for variables from X_1 will be chosen by Player 1 via his choice of letters in the width game. Gadgets will then be added to

- control whether the current valuation makes the formula φ true,
- set the initial valuation to α_{init} and
- make the automaton universal.

The width game should be lost immediately by the player who chose a valuation making the formula false.

These components will first be built as an auxiliary automaton \mathcal{B} . In order to properly mimic the dynamic of the game \mathcal{G}_c , one needs to additionally constrain the letters that can be played by Player 1. This will be done by a separate deterministic automaton \mathcal{C} , forcing the word played by Player 1 to belong to a certain language. Finally, \mathcal{A} will be obtained by a cartesian product of \mathcal{B} and \mathcal{C} . In order to illustrate the construction of \mathcal{B} and \mathcal{C} , we will use a running example:

Example 6.3. The instance I_c of \mathcal{G}_c we take as example is $(\varphi, X_0, X_1, \alpha_{init})$, with $\varphi = (x \vee y \vee z \vee t) \wedge (\bar{x} \vee y \vee \bar{z} \vee \bar{t})$, $X_0 = \{x, y\}$, $X_1 = \{z\}$, and α_{init} to be the valuation setting all variables to true. Notice that this instance is won by Player 0, as it suffices to always set y to true to guarantee that φ remains true.

After defining \mathcal{B} and \mathcal{C} in Sections 6.2.2 and 6.2.3, we will prove in Section 6.3 that the game $\mathcal{G}_w(\mathcal{A}, k)$ correctly emulates the game \mathcal{G}_c on instance I_c .

6.2.2. *The automaton \mathcal{B} .* The automaton \mathcal{B} is the main gadget of the construction. The idea is that moving k pebbles in \mathcal{B} will be equivalent to choosing a valuation of k variables, via a set of $2k$ states (called Q_{Lit}): one state for every literal, corresponding to a truth value assignment for its associated variable. The transitions of \mathcal{B} are designed so that for variables in X_0 , Player 0 can choose the valuation using the nondeterminism of \mathcal{B} on a single letter a , while for variables in X_1 , Player 1 chooses a valuation by choosing which letters of the form f_l to play.

Let us start by describing the automaton \mathcal{B} associated to the running example 6.3.

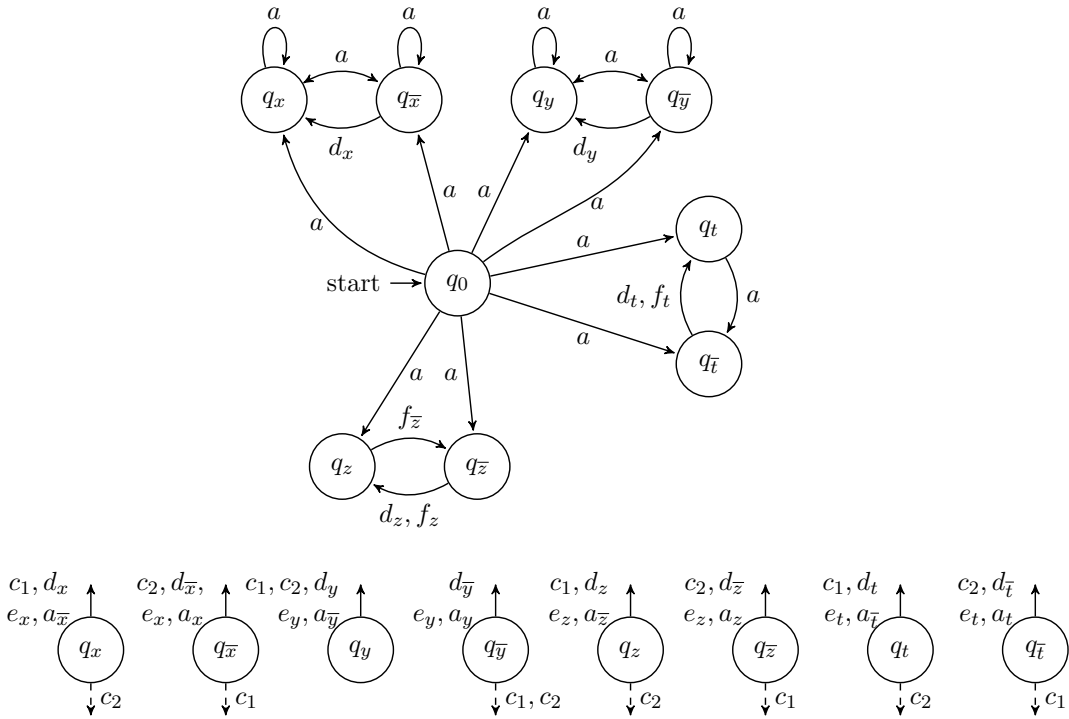


Figure 3: Valuation gadget and exit transitions of the automaton \mathcal{B}

Example 6.4. The automaton \mathcal{B} corresponding to Example 6.3 is described in Figure 3. The first diagram represents initial transitions and transitions changing the valuation. Deterministic self-loops such as $q_x \xrightarrow{d_x} q_x$ are omitted for readability. The second diagram describes for each state q_l which letters from $\{c_1, c_2\}$ cannot be read (dashed arrows to the bottom), and which letters go to the accepting sink q_T (arrows to the top). The automaton \mathcal{B} is safe, i.e. all states are accepting. The only way for a run to fail is to read a letter c_i in a state where it is forbidden. These letters are used to control that the valuation chosen by Player 0 makes the formula true. For instance reading c_2 in q_x leads to a fail of the run, because x does not make clause 2 true in the formula φ . If Player 1 can play a letter c_i such that no pebble is in a state q_l where l makes clause c_i true, he immediately wins the game.

In this example, we can notice that the letter a is used to allow Player 0 to set the values of variables x and y via nondeterminism. On the other hand, the letters f_z and $f_{\bar{z}}$ can be played by Player 1 to set the value of variable z . Letters d_l for each literal l have

two roles: they set the initial valuation (here with all variables to true), and they will help to guarantee that the final automaton is universal, by immediately leading to an accepting state if letter d_l is read in state q_l .

We now give the formal definition of the general construction of $\mathcal{B} = (Q, \Sigma, q_0, \Delta, F)$.

States

Let $Q_{Lit} = \{q_l \mid l \in Lit\}$. The states in Q_{Lit} will be used to encode valuations α , via the positions of the pebbles in the game $\mathcal{G}_w(\mathcal{A}, k)$.

Q_{Lit} is partitioned into Q_0, Q_1, Q_t , with $Q_i = \{q_l \mid l \in Lit_i\}$ for $i \in \{0, 1\}$ and $Q_t = \{t, \bar{t}\}$.

We finally set $Q = \{q_0, q_\top\} \cup Q_{Lit}$ where q_0 is the initial state and q_\top is an accepting sink state. Reaching q_\top with one of its pebble will mean immediate win for Player 0, as this pebble will trivially accept any word subsequently played by Player 1. We define $F = Q$, i.e. \mathcal{B} is a safety automaton, and every run is accepting. Notice that the number of states of \mathcal{B} is $|Q| = 2 + 2k$, so it is polynomial in the size of the instance I_c of \mathcal{G}_c .

Alphabet

We define here several sub-alphabets that will be used in our encoding. For each of them, we already give an intuition of how it will be used. The alphabet is presented in two groups, separated by a double line: the first group will be used in the normal flow of the simulation of the game, while the second group is used by Player 1 to challenge choices of Player 0, and will normally never be played if Player 0 is playing a correct strategy.

$\{a, f_t\}$	The letter a allows Player 0 to choose a value for all variables from X_0 , and sets variable t to false. The letter f_t is used to set variable t to true.
$\Gamma_{Lit} := \{a_l \mid l \in Lit\}$	For each clause, Player 1 will have to play a letter a_l witnessing that his valuation makes this clause true.
$\Gamma_1 := \{f_l \mid l \in Lit_1\}$	The letter f_l will be played by Player 1 to set the literal l to true.
$\Sigma_D := \{d_l \mid l \in Lit\}$	Used just once at the beginning: sets initial valuation while making the automaton universal.
$\Sigma_C := \{c_i \mid i \in [1, n]\}$	A letter c_i will be played by Player 1 if the valuation chosen by Player 0 fails to make the clause C_i true.
$\Sigma_V := \{e_v \mid v \in V\}$	The letter e_v will be played by Player 1 if Player 0 has failed to set a value for variable v .

We set $\Sigma = \{a, f_t\} \cup \Gamma_{Lit} \cup \Gamma_1 \cup \Sigma_D \cup \Sigma_C \cup \Sigma_V$.

Notice that $|\Sigma| \leq 2 + 2k + 2k + 2k + n + k = n + 7k + 2$, so it is polynomial in the size of I_c as well.

Transitions

We will use the notation $p \xrightarrow{a} q$ to mean that we put a transition (p, a, q) in Δ . If $l \in Lit$ is a literal, we define its projection $\pi(l)$ to variables by $\pi(l) = v$ if $l \in \{v, \bar{v}\}$. We define the negation of a literal $l \in \{v, \bar{v}\}$ as $\bar{l} = \bar{v}$ if $l = v$ and $\bar{l} = v$ if $l = \bar{v}$.

We present the transition table Δ in the following array: the left column contains the set of transitions, while the right column explains their roles in the reduction. As before, we separate in a second component transitions that will only be taken in the event of a Player failing to play a valid strategy, thereby terminating the simulation of the game.

$q_0 \xrightarrow{a} q_l$ for all $l \in Lit$	allows Player 0 to choose any initial valuation.
$q_l \xrightarrow{d_{l'}} q_{l''}$ if $l \neq l'$ and $l'' = \alpha_{init}(\pi(l))$	sets the initial valuation to α_{init} .
$q_l \xrightarrow{f_{l'}} q_{l'}$ if $\pi(l) = \pi(l')$, for all $l' \in Lit_1 \cup \{t\}$	sets the value of $\pi(l)$ to l' .
$q_l \xrightarrow{f_{l'}} q_l$ if $\pi(l) \neq \pi(l')$, for all $l' \in Lit_1 \cup \{t\}$	leaves other variable unchanged.
$q_l \xrightarrow{a} q_{l'}$ if $l \in Lit_0$ and $\pi(l) = \pi(l')$, or if $l \in Lit_1$ and $l = l'$	nondeterministic choice for variables in X_0 , leaving variables from X_1 unchanged.
$q_l \xrightarrow{a} q_{\bar{t}}$ if $\pi(l) = t$	sets t to false during the turn of Player 0.
$q_l \xrightarrow{a_l} q_l$ if $l \in Lit$	will be used to validate the valuation chosen by Player 1.
$q_l \xrightarrow{d_l} q_{\top}$ for all $l \in Lit$	helps towards universality of the automaton: a good guess from Player 0 on which d_l will be played leads to immediate acceptance.
$q_l \xrightarrow{e_{\pi(l)}} q_{\top}$ for all $l \in Lit$	e_v is played if no value is set for variable v , leading to instant loss for Player 0 if no value is set and instant win otherwise.
$q_l \xrightarrow{c_i} q_{\top}$ if literal l appears in C_i	if Player 1 challenges the valuation with clause C_i , instant win for Player 0 if a literal makes the clause true.
$q_l \xrightarrow{a_{\bar{t}}} q_{\top}$ if $l \in Lit$	if Player 1 tries to validate his valuation with a wrong literal, instant win for Player 0.
$q_{\top} \xrightarrow{b} q_{\top}$ for all $b \in \Sigma$	accepting sink state.

This achieves the definition of \mathcal{B} . Notice that \mathcal{B} has size polynomial in the size of I_c , and can be computed from I_c in polynomial time.

6.2.3. *The automaton \mathcal{C} .* The automaton \mathcal{C} is used to restrict the moves of Player 1, i.e. the letters chosen in the width game, to those that are relevant to the game \mathcal{G}_c . This includes for instance forcing him to prove that his own valuations make the formula true via the letters a_l , and allowing him to challenge valuations chosen by Player 0 via the letters c_i . We define a safety language such that if Player 1 plays a bad prefix of this language, then the whole automaton \mathcal{A} immediately goes to an accepting sink state, and therefore Player 0 wins the width game.

We formally describe \mathcal{C} in the following, and instantiate it on the running example.

For each $i \in [1, n]$, let $A_i = \{a_l \mid \text{literal } l \text{ appears in } C_i\}$.

Let $L_{val} = A_1 A_2 \dots A_n$, L_{val} is a subset of $(\Gamma_{Lit})^n$. The purpose of L_{val} is to check that the valuation chosen by Player 1 (corresponding to Player 1 in \mathcal{G}_c) is valid, by forcing him to choose one valid literal by clause.

We define

$$L_C = a_{\Sigma_D}(\varepsilon + \Sigma_V)(\Gamma_1^{|X_1|} \cdot f_t \cdot L_{val} \cdot a(\varepsilon + \Sigma_V)(\varepsilon + \Sigma_C))^*.$$

We detail in the following the meaning of the different factors in this expression, by order of appearance:

$a\Sigma_D$	sets the initial configuration, and makes the automaton accept anything
$(\varepsilon + \Sigma_V)$	punishes Player 0 if a variable is not instantiated
$\Gamma_1^{ X_1 } \cdot f_t$	chooses values for variables in X_1 and sets t to true
L_{val}	proves that the chosen valuation makes φ true
a	allows Player 0 to choose a valuation for X_0 , and sets t to false
$(\varepsilon + \Sigma_V)$	verifies that each variable is still instantiated (x and \bar{x} could be set to true while erasing the valuation of y).
$(\varepsilon + \Sigma_C)$	allows to challenge the valuation with a clause C_i .

Let \mathcal{C}_0 be a complete DFA recognizing L_C with rejecting sink state \top_C . We build \mathcal{C} by making all states of \mathcal{C}_0 accepting, including its sink state \top_C . This makes \mathcal{C} a universal complete DFA, and any word that is not a prefix of a word in L_C will reach the accepting sink state \top_C . It is straightforward to build \mathcal{C} in polynomial time from I_c , with the size of \mathcal{C} polynomial in the size of I_c .

Example 6.5. The deterministic safety automaton \mathcal{C} corresponding to the running example 6.3 is represented in Figure 4. Some states are labeled by the type of the position they represent, see next section. Transitions to the accepting sink \top_C are not represented: all missing transitions for this deterministic automaton to be complete go to state \top_C .

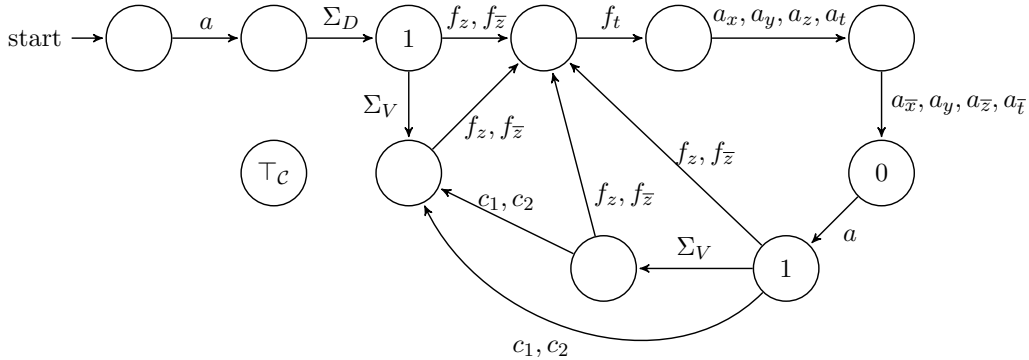


Figure 4: The complete safety DFA \mathcal{C}

6.2.4. *The main automaton \mathcal{A} .* We now combine \mathcal{B} and \mathcal{C} to obtain the automaton \mathcal{A} , for which being able to compute width will amount to solve the game \mathcal{G}_c .

The automaton \mathcal{A} is defined as the cartesian product of \mathcal{B} and \mathcal{C} , with the additional modification that all states of the form (q, \top_C) or (q_\top, q') are merged to a unique accepting sink state $\top_{\mathcal{A}}$. We also add all transitions of the form $(q, q') \xrightarrow{x} \top_{\mathcal{A}}$ as soon as $q' \xrightarrow{x} \top_C$ is a transition of \mathcal{C} , even when the letter x cannot be read from q in \mathcal{B} .

Since \mathcal{C} is deterministic, it has no impact on the width, and the only non-determinism for Player 0 to resolve in $\mathcal{G}_w(\mathcal{A}, k)$ comes from \mathcal{B} .

Lemma 6.6. *\mathcal{A} is a universal safety automaton.*

Proof. Notice that since \mathcal{B} and \mathcal{C} are safety automata, \mathcal{A} is a safety automaton as well. We have to show that \mathcal{A} accepts all words.

Let $w \in \Sigma^*$. If w truncated to its first two letters is not a prefix of ad_l for some $l \in Lit$, then it immediately ends up in state $\top_{\mathcal{C}}$ in \mathcal{C} , and so it reaches $\top_{\mathcal{A}}$ in \mathcal{A} . So in this case, $w \in L(\mathcal{A})$. Assume now that $w = ad_lv$ for some $l \in Lit$ and $v \in \Sigma^*$.

Then the run $q_0 \xrightarrow{a} q_l \xrightarrow{d_l} q_{\top} \xrightarrow{v} q_{\top}$ of \mathcal{B} is a witness that \mathcal{A} can reach $\top_{\mathcal{A}}$ when reading w .

Finally, the words ε and a are also accepted by \mathcal{A} .

Therefore, any $w \in \Sigma^*$ is accepted by \mathcal{A} , and \mathcal{A} is a universal safety automaton. \square

We are now ready to prove that the construction of \mathcal{A} performs the wanted reduction, thereby completing the proof of Theorem 6.1.

6.3. Proof of correctness. We prove in this section that the above construction allows to use the width game of \mathcal{A} to emulate the game \mathcal{G}_c .

Let $k = |V|$, we want to prove that $\text{width}(\mathcal{A}) > k$ if and only if Player 1 wins in I_c .

We will show that the game $\mathcal{G}_w(\mathcal{A}, k)$ simulates the game \mathcal{G}_c , by establishing a correspondence between strategies for these two games.

Player 1 wins in $I_c \implies \text{width}(\mathcal{A}) > k$.

Assume Player 1 wins in I_c , with a winning strategy σ_c . We aim at building a winning strategy σ_w for Player 1 in $\mathcal{G}_w(\mathcal{A}, k)$. It means that Player 1 can enforce a position of the game where the word played is in $L(\mathcal{A}) = \Sigma^*$, but all pebbles have been erased due to non-existing transitions.

We now define σ_w . Following the language L_C , Player 1 starts by playing a . Player 0 can move the pebbles to any subset of Q_{Lit} of size at most k . In order to prevent Player 0 from reaching $\top_{\mathcal{A}}$, Player 1 will now play d_l where q_l is a state not occupied by a pebble. This puts all pebbles to the states corresponding to the initial valuation.

If not all variables are instantiated by a pebble, Player 1 plays $e_v \in \Sigma_v$ such that q_v and $q_{\bar{v}}$ do not contain a pebble. The resulting word ad_1e_v is in $L(\mathcal{A}) = \Sigma^*$, but Player 0 fails to accept it, since no state occupied by a pebble can read e_v , so Player 1 wins the game. We can therefore assume that Player 0 uses all k pebbles and reaches all states q_l corresponding to the valuation α_{init} . In this case we define strategy σ_w so that Player 1 does not play a letter in Σ_V , as allowed by the ε in the definition of L_C .

We now switch to the main dynamic of the game, and we will match certain positions of $\mathcal{G}_w(\mathcal{A}, k)$ to positions of the game I_c . The current position of $\mathcal{G}_w(\mathcal{A}, k)$, where $a \cdot d_l$ has been played and the k pebbles are in the states of Q_{Lit} matching α_{init} , corresponds to the initial position $(1, \alpha_{init})$ of I_c .

More generally the positions reached after playing a word from L_C will be matched to positions $(1, \alpha)$ of I_c , where α is described by the states q_l occupied by the k pebbles. We say that such a position of $\mathcal{G}_w(\mathcal{A}, k)$ is of type 1. Similarly the positions reached after playing a word from $L_C(\Gamma_1^{|X_1|} \cdot f_t \cdot L_{val})$ will be matched to a position $(0, \alpha')$ of I_c , and are called positions of type 0.

We define σ_w in positions of type 1 in the following way: Let α_1 be the values of variables in X_1 chosen by σ_c in the matching position of I_c . The factor of $\Gamma_1^{|X_1|}$ played by σ_w will explicit these values, by playing for each literal l that is true in α_1 the letter f_l . This switches

the pebbles in X_1 to match the valuation α_1 , and leave the other variables (from $X_0 \cup \{t\}$) unchanged.

The letter f_t must then be played by Player 1, in order to avoid losing by allowing Player 0 to put his pebbles in $\top_{\mathcal{A}}$ (any other letter would lead the deterministic run of \mathcal{C} to $\top_{\mathcal{C}}$). This moves the t pebble to q_t , setting the value of t to 1, according to the definition of \mathcal{G}_c .

Player 1 must now play a word in L_{val} . Since σ_c is winning, the current valuation α makes the formula true. The strategy σ_w consists in witnessing this by choosing for each clause C_i a literal l from α that makes it true, and play a_l . This leaves the position of the pebbles unchanged.

We have now reached a position of type 0. We define the strategy σ_w for these positions. First, Player 1 must play the letter a . This allows Player 0 to move pebbles from Q_0 freely, thereby choosing a new valuation for variables in X_0 . Moreover it moves a pebble from q_t to $q_{\bar{t}}$, setting the value of t to 0. Notice that by the definition of $\mathcal{G}_w(\mathcal{A}, k)$, Player 0 could also duplicate some pebbles and erase some others, thereby setting some variables in X_0 to both true and false, and not assigning other variables from V . If Player 0 chooses to do this, the strategy σ_w of Player 1 will immediately punish it by playing e_v where v is a non-assigned variable, and as before this allows Player 1 to win the game. This allows us to continue assuming the pebbles describe a valuation α of all variables in V . If α makes the formula true, we are back to a position of type 1, and we continue with the strategy as described.

On the other hand, if we have reached a winning position for Player 1 in I_c , i.e. if the valuation α makes the formula false, we show that Player 1 can win $\mathcal{G}_w(\mathcal{A}, k)$. To do so, he plays a letter c_i such that no literal in C_i is true in α . This way, no pebble is in a state where c_i can be read, and no pebbles are present in the next position of $\mathcal{G}_w(\mathcal{A}, k)$. Since the word w played until now is in the language $L(\mathcal{A}) = \Sigma^*$, this is a winning position for Player 1 in $\mathcal{G}_w(\mathcal{A}, k)$.

Since σ_c is winning, the game will eventually reach a position where the valuation chosen by Player 0 makes the formula φ false, hence σ_w is a winning strategy for Player 1 in $\mathcal{G}_w(\mathcal{A}, k)$, witnessing $\text{width}(\mathcal{A}) > k$.

Player 0 wins in $I_c \implies \text{width}(\mathcal{A}) \leq k$.

Assume that Player 0 has a winning strategy σ_0 in I_c . This means that this strategy avoids losing positions for Player 0, either by playing forever, or by reaching a position that is losing for Player 1. Moreover, since \mathcal{G}_c is a safety game for Player 0, σ_0 can be chosen positional, i.e. its move only depends on the current valuation α .

We will show that this strategy σ_0 can be turned into a strategy τ_0 that is winning for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$, thereby witnessing $\text{width}(\mathcal{A}) \leq k$.

The strategy τ_0 consists in the following:

- on the first occurrence of a , put the pebbles in all states q_l with l appearing in α_{init} (or to any other valuation),
- on other occurrences of a , match the choice of σ_0 for the valuation of X_0 (according to the current valuation α).

Other choices to be made by τ_0 are described in the following.

First of all, we may assume that Player 1 always plays words from $L_{\mathcal{C}}$, otherwise he immediately loses $\mathcal{G}_w(\mathcal{A}, k)$, as the automaton \mathcal{A} goes to $\top_{\mathcal{A}}$.

After the prefix from $a\Sigma_D$, Player 1 has no interest in playing a letter in Σ_V , since the strategy τ_0 assigned a value to each variable.

We are then in a position of type 1, and Player 1 must play a word in $\Gamma_1^{|X_1|}$. This allows him to choose a valuation for any variable in X_1 . The letter f_t then sets the value of t to 1, according to the rules of \mathcal{G}_c .

Player 1 must now play a word from L_{val} . We show that this forces him to prove that the current valuation α makes the formula φ true. Indeed, for each clause i , Player 1 must choose a literal l appearing in C_i , and play a_l . If l is currently false, i.e. there is a pebble in $q_{\bar{l}}$, Player 0 can move this pebble to q_{\top} and wins the game $\mathcal{G}_w(\mathcal{A}, k)$. Therefore, if Player 1 cannot choose a valuation of X_1 setting φ to true, Player 0 wins the game $\mathcal{G}_w(\mathcal{A}, k)$.

Otherwise, we reach a position of type 0. Letter a allows Player 0 to choose a valuation for X_0 , and sets t to 0. Strategy τ_0 is defined to do this accordingly to σ_0 , and therefore this will always reach a valuation α setting φ to true. If Player 1 plays a letter from Σ_V , Player 0 can reach q_{\top} and win the game. If Player 1 plays a letter c_i from Σ_C , it will allow Player 0 to reach q_{\top} , since one literal l of clause C_i is currently set to true, via a pebble in q_l . Therefore, the only interesting move for Player 1 is to go back to his move in a position of type 1, and play a new valuation of X_1 via a word in $\Gamma_1^{|X_1|}$.

Since σ_0 is winning in I_c , either the play goes on forever in I_c , which means Player 0 wins the corresponding play in $\mathcal{G}_w(\mathcal{A}, k)$, or Player 0 is eventually able to reach q_{\top} , either because Player 1 loses in I_c via a bad valuation, or because he made bad choices in $\mathcal{G}_w(\mathcal{A}, k)$, for instance by playing a letter from Σ_V . Either way, the strategy σ_0 is winning for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$.

This achieves the proof that $\text{width}(\mathcal{A}) > k$ if and only if Player 1 wins in I_c . Since the reduction from an instance of I_c to an instance of the width problem for a universal safety NFA can be done in polynomial time, we showed the following theorem:

Theorem 6.7. *Computing the width of a universal safety NFA is EXPTIME-complete.*

By Corollary 5.5, we obtain the following result:

Corollary 6.8. *It is EXPTIME-complete to decide, given two NFAs \mathcal{A}, \mathcal{B} and $k \geq 1$, whether $\mathcal{A} \sqsubseteq_k \mathcal{B}$. This is already true when \mathcal{A} is fixed to the trivial automaton $\mathcal{A}_{\text{triv}}$ and the input \mathcal{B} is restricted to universal safety NFAs.*

Remark 6.9. Notice that this proof also shows that the alternative versions of width and multipbble simulations where pebbles cannot be duplicated are also EXPTIME-complete. This is because our reduction actually only needs this kind of moves, and uses the Σ_V gadget to forbid duplicating pebbles while erasing others. Moreover, all results from Section 5.2 can be carried to this alternative version.

Remark 6.10. Although the present section deals with finite words, most results are immediately transferable to safety automata on infinite words. Any infinite run is accepting in a safety automaton. This acceptance condition is of particular interest in verification, as it describes very natural properties such as deadlock freeness of a system. See [3] for an introduction to automata on infinite words for verification. This also shows that the EXPTIME-hardness result can be lifted to any acceptance condition generalising the safety one, such as Büchi, coBüchi, parity.

7. COBÜCHI AUTOMATA

We now turn to the case of coBüchi automata, and their determinisation problem. Here, since GFG and DBP are no longer equivalent [4, 20], aiming for a GFG automaton becomes a problem that is different from determinization via DBP automata. We will compare these two problems, and we will see that the class of GFG coBüchi automata is particularly interesting for several reasons.

First of all recall that NCA and DCA have same expressive power, i.e. the determinisation of coBüchi automata does not need to introduce more complex acceptance conditions. This follows from the breakpoint construction [25] that we will generalise in this section to its incremental variant.

7.1. Width of ω -automata. We define here the width of automata on infinite words in a general way, as the definition is independent of the accepting condition.

Let $\mathcal{A} = (Q, \Sigma, q_0, \Delta, \alpha)$ be an automaton on infinite words with acceptance condition α , and $n = |Q|$ be the size of \mathcal{A} .

As before, we want to define the *width* of a \mathcal{A} as the minimum number of states that need to be tracked in order to deterministically build an accepting run in an online way.

We will use the same family of games $\mathcal{G}_w(\mathcal{A}, k)$ as in Section 3.1, and they will only differ in the winning condition.

The game $\mathcal{G}_w(\mathcal{A}, k)$ is played on $Q^{\leq k}$, starts in $X_0 = \{q_0\}$, and the round i of the game from a position $X_i \in Q^{\leq k}$ is defined as follows:

- Player 1 chooses a letter $a_{i+1} \in \Sigma$.
- Player 0 moves to a subset $X_{i+1} \subseteq \Delta(X_i, a_{i+1})$ of size at most k .

An infinite play is winning for Player 0 if whenever $a_1 a_2 \dots \in L(\mathcal{A})$, the sequence $X_0 X_1 X_2 \dots$ contains an accepting run. That is to say there is a valid accepting run $q_0 q_1 q_2 \dots$ of \mathcal{A} on $a_1 a_2 \dots$ such that for all $i \in \mathbb{N}$, $q_i \in X_i$.

Definition 7.1. The width of \mathcal{A} , denoted $\text{width}(\mathcal{A})$, is the least k such that Player 0 wins $\mathcal{G}_w(\mathcal{A}, k)$.

As before, an automaton \mathcal{A} is GFG if and only if $\text{width}(\mathcal{A}) = 1$.

7.2. GFG coBüchi automata. We recall here previous results on GFG coBüchi automata.

The first result is the exponential succinctness of GFG NCAs compared to DCAs.

Theorem 7.2 Theorem 1, [20]. *There is a family of languages $(L_n)_{n \in \mathbb{N}}$ such that for all n , L_n is accepted by a coBüchi GFG automaton of size n , but any deterministic parity automaton for L_n must have size in $\Omega\left(\frac{2^n}{n}\right)$.*

Moreover, each language L_n can be chosen LTL-definable [17], hinting towards applicability of GFG NCAs in LTL synthesis.

This means that some GFG NCA only admit GFG strategies with exponential memories [20], i.e. the witness for an NCA being GFG can be of exponential size. Despite this fact, the next theorem shows that GFG NCAs can be recognised efficiently.

Theorem 7.3 Theorem 11, [20]. *Given an NCA \mathcal{A} , it is in PTIME to decide whether \mathcal{A} is GFG.*

It was also shown that GFG coBüchi automata can be efficiently minimized:

Theorem 7.4 Theorem 21, [28]. *Cobüchi automata with acceptance condition on transitions can be minimized in polynomial time.*

The conjunction of these results make the coBüchi class particularly interesting in our setting: the succinctness allows us to potentially save a lot of space compared to classical determinisation, and Theorem 7.3 can be used to stop the incremental construction. Moreover, once a coBüchi GFG automata has been built, it can be minimized efficiently thanks to Theorem 7.4. Since GFG automata suffice for many purposes, for instance in a context where we want to test for inclusion, or compose the automaton with a game, this makes the class of GFG coBüchi automata particularly interesting.

We examine later the case where GFG automata are not enough and we are aiming at building a DCA instead.

7.3. Partial breakpoint construction. We generalise here the breakpoint construction from [25], in the same spirit as Section 3.2.

Let us first briefly recall the breakpoint construction. If $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$ is an NCA, then a state of its determinized automaton is of the form (X, Y) , with $Y \subseteq X$. The powerset construction is performed on both sets, but the Y -component deletes rejecting states. The new transition function δ is defined as $\delta(X, Y) = (\Delta(X), \Delta(Y) \cap F)$, if $Y \neq \emptyset$. States with an empty second component are “breakpoints”: they are rejecting, but they allow to reset the second component to the first one: $\delta(X, \emptyset) = (\Delta(X), \Delta(X))$. The resulting deterministic run will accept if and only if the second component is eventually non-empty, witnessing the existence of an accepting run in \mathcal{A} .

We now describe the incremental version of this construction. For a parameter k , we want the k -breakpoint construction to be able to keep track of at most k states simultaneously.

Given an NCA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, we define the k -breakpoint construction of \mathcal{A} as the NCA $\mathcal{A}_k = (Q', \Sigma, \Delta', (\{q_0\}, \{q_0\}), F')$, with

$$Q' = \{(X, Y) \mid X, Y \in Q^{\leq k} \text{ and } Y \subseteq X\},$$

$$\Delta'((X, Y), a) := \begin{cases} \{(\Delta(X, a), \Delta(X, a))\} & \text{if } Y = \emptyset \text{ and } |\Delta(X, a)| \leq k \\ \{(X', X') \mid X' \subseteq \Delta(X, a), |X'| = k\} & \text{if } Y = \emptyset \text{ and } |\Delta(X, a)| > k \\ \{(\Delta(X, a), \Delta(Y, a) \cap F)\} & \text{if } Y \neq \emptyset \text{ and } |\Delta(X, a)| \leq k \\ \{(X', X' \cap (\Delta(Y, a) \cap F)) \mid X' \subseteq \Delta(X, a), |X'| = k\} & \text{otherwise} \end{cases}$$

$$F' := \{(X, Y) \in Q' \mid Y \neq \emptyset\}$$

That is, a run is accepting in \mathcal{A}_k if it visits the states of the form (X, \emptyset) finitely many times.

Lemma 7.5. *The number of states of \mathcal{A}_k is at most $\sum_{i=0}^k \binom{n}{i} 2^i$, which is in $O\left(\frac{(2n)^k}{k!}\right)$.*

Proof. A state of \mathcal{A}_k is of the form (X, Y) with $|X| \leq k$ and $Y \subseteq X$. Therefore, there are at most $\sum_{i=0}^k \binom{n}{i} 2^i$ such states. Since $\binom{n}{i} \leq \frac{n^i}{i!}$, we can bound the number of states by $\sum_{i=0}^k \frac{n^i}{i!} 2^i \leq \frac{n^k}{k!} 2^{k+1} = O\left(\frac{(2n)^k}{k!}\right)$ \square

Lemma 7.6. *$L(\mathcal{A}) = L(\mathcal{A}_k)$, and $\text{width}(\mathcal{A}) \leq k \iff \mathcal{A}_k$ is GFG.*

Proof. First let us show that $L(\mathcal{A}) = L(\mathcal{A}_k)$.

Let $w \in L(\mathcal{A})$, witnessed by an accepting run $\rho = q_0 q_1 q_2 \dots$. The run ρ can be used to resolve the nondeterminism of \mathcal{A}_k while reading w , by choosing at each step i as first

component any set X containing q_i . Since for all i big enough, $q_i \in F$, after this point there is at most one empty second component in the run of \mathcal{A}_k , and therefore $w \in L(\mathcal{A}_k)$.

Conversely, let $w \in L(\mathcal{A}_k)$, witnessed by an accepting run $\rho = (X_0, Y_0)(X_1, Y_1) \dots$. We consider the DCA \mathcal{D} obtained from \mathcal{A} via the breakpoint construction [25]. The states of \mathcal{D} are of the form (X, Y) with $X \supseteq Y$, and its acceptance condition is the same as the one of \mathcal{A}_k . Let $\rho' = (X'_0, Y'_0)(X'_1, Y'_1) \dots$ be the run of \mathcal{D} on w . By definition of \mathcal{A}_k , for all $i \in \mathbb{N}$ we have $X_i \subseteq X'_i$ and $Y_i \subseteq Y'_i$. Since Y_i is empty for finitely many i , it is also the case for Y'_i , and therefore ρ' is accepting. We obtain $w \in L(\mathcal{D}) = L(\mathcal{A})$.

Now we shall show $\text{width}(\mathcal{A}) \leq k \implies \mathcal{A}_k$ is GFG.

Assume that there is a winning strategy $\sigma_w : \Sigma^* \rightarrow Q^{\leq k}$ for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$. We show that this induces a GFG strategy $\sigma_{GFG} : \Sigma^* \rightarrow Q'$ for \mathcal{A}_k . First, notice that without loss of generality, we can assume that for any $(u, a) \in \Sigma^* \times \Sigma$ such that $|\Delta(\sigma_w(u), a)| \leq k$, we have $\sigma_w(ua) = \Delta(\sigma_w(u), a)$. Indeed, it is always better for Player 0 to choose a set as big as possible.

Using this assumption, the strategy σ_{GFG} is naturally defined from σ_w by relying on the first component, i.e. $\sigma_{GFG}(u) := (X', Y')$, where $X' = \sigma_w(u)$, and Y' is forced by the transition table of \mathcal{A}_k . That is, if $Y \neq \emptyset$ we have $Y' = X' \cap \Delta(Y, a) \cap F$, and else ($Y = \emptyset$) we have $Y' = X'$. We show that σ_{GFG} is indeed a GFG strategy. Let w be an infinite word in $L(\mathcal{A}_k) = L(\mathcal{A})$. We must show that the run $\rho_w = (X_0, Y_0)(X_1, Y_1)(X_2, Y_2) \dots$ of \mathcal{A}_k induced by σ_{GFG} on w is accepting. Since σ_w is a winning strategy in $\mathcal{G}_w(\mathcal{A}, k)$, there is an accepting run $\rho = q_0q_1q_2 \dots$ of \mathcal{A} such that for all $i \in \mathbb{N}$, $q_i \in X_i$. This means that there is $N \in \mathbb{N}$ such that for all $i \geq N$, $q_i \in F$. If for all $i \geq N$, $Y_i \neq \emptyset$, the run ρ_w is accepting. Otherwise, let $M > N$ be such that $Y_M = \emptyset$. By definition of \mathcal{A}_k , we get $Y_{M+1} = X_{M+1}$. For $i \geq M + 1$, we will always have $q_i \in Y_i$, by definition of \mathcal{A}_k , therefore $Y_i \neq \emptyset$. We can conclude that ρ_w is accepting, and therefore \mathcal{A}_k is GFG.

It remains to prove that \mathcal{A}_k is GFG $\implies \text{width}(\mathcal{A}) \leq k$.

Let $\sigma_{GFG} : \Sigma^* \rightarrow Q'$ be a GFG strategy for \mathcal{A}_k . We build a strategy $\sigma_w : \Sigma^* \rightarrow Q^{\leq k}$ for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$, witnessing that $\text{width}(\mathcal{A}) \leq k$.

For all $u \in \Sigma^*$, we define $\sigma_w(u)$ to be the first component of $\sigma_{GFG}(u)$. Let $w \in L(\mathcal{A}) = L(\mathcal{A}_k)$, so the run $\rho = (X_0, Y_0)(X_1, Y_1)(X_2, Y_2) \dots$ induced by σ_{GFG} on w is accepting. We have to show that the play $\pi = X_0X_1X_2 \dots$ is winning for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$, i.e. that there exists an accepting run $\rho_\pi = q_0q_1q_2 \dots$ of \mathcal{A} with $q_i \in X_i$ for all $i \in \mathbb{N}$. Assume no such run exists, i.e. all runs included in π are rejecting. Let $N \in \mathbb{N}$ be such that for all $i \geq N$, $Y_i \neq \emptyset$. Any run included in π and starting in $q \in Y_N$ must encounter a non-accepting state. This means that there is $K > N$ such that between indices N and K , every run included in π contains a non-accepting state. By definition of \mathcal{A}_k , this implies there is Y_i with $i > N$ such that $Y_i = \emptyset$. This contradicts the definition of N , therefore there must be an accepting run ρ_π included in π . \square

7.4. Incremental construction of GFG NCA. Suppose we are given an NCA \mathcal{A} , and we want to build an equivalent GFG automaton.

We can do the same as in Section 3.4: incrementally increase k and test whether \mathcal{A}_k is GFG, which is in PTIME by Theorem 7.3. However in the coBüchi setting, the GFG automaton is not necessarily DBP, and can actually be more succinct than any deterministic automaton for the language (Theorem 7.2).

If we are in a context where we are satisfied with a GFG automaton, such as synthesis or inclusion testing, this procedure can provide us one much more efficiently than determinisation.

Indeed, the example from Theorem 7.2 showing that GFG NCA are exponentially succinct compared to deterministic automata can be easily generalised to any width. For instance if our procedure is applied to the product of the GFG NCA of size n from Theorem 7.2 with the one from Example 3.8, our construction will stop at the second step and generate a GFG automaton of quadratic size. This shows that the incremental construction for finding an equivalent GFG NCA can be very efficient compared to determinisation.

7.5. Complexity of the width problem for NCAs. As stated in Remark 6.10, directly computing the width of an NCA is EXPTIME-hard. The above construction together with Lemma 7.6 gives an EXPTIME algorithm solving the width problem for an input \mathcal{A}, k with \mathcal{A} an NCA: build \mathcal{A}_k , and test whether it is GFG in polynomial time. This shows that the EXPTIME-completeness of the width problem can be lifted to the coBüchi condition.

Corollary 7.7. *The width problem for NCAs is EXPTIME-complete.*

7.6. Aiming for determinism. In cases where a GFG automaton is not enough, and we want instead to build a DCA, we can test whether the current automaton is DBP instead of GFG in the incremental algorithm. If we find the automaton is DBP, we can remove the useless transitions, and obtain an equivalent DCA. This procedure will always stop, as in the worst case it will eventually reach the breakpoint construction, which directly builds a DCA.

Notice that the number of steps in this procedure corresponds to an alternative notion of width that can be called *det-width*. The det-width of an automaton \mathcal{A} is the least k such that Player 0 has a positional winning strategy in $\mathcal{G}_w(\mathcal{A}, k)$. Det-width always matches width on finite words by Theorem 3.7, but the notions diverge on infinite words.

This section studies the complexity of checking whether an NCA is DBP. The next theorem shows that surprisingly, this is harder to check than being GFG for NCAs.

Theorem 7.8. *Given an NCA \mathcal{A} , it is NP-complete to check whether it is DBP.*

We first show the hardness with the following lemma.

Lemma 7.9. *Checking whether an NCA is DBP is NP-hard.*

Proof. We prove this by reduction from the Hamiltonian Cycle problem on a directed graph, which is known to be NP-complete [15].

Recall that a Hamiltonian cycle is a cycle using each vertex of the graph exactly once.

Suppose we have a directed graph $G = ([1, n], E)$ and we want to check whether it contains a Hamiltonian cycle. W.l.o.g. we can assume that the graph is strongly connected, otherwise the answer is trivially no.

We construct an NCA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$, where F is the set of accepting states, such that \mathcal{A} is DBP if and only if G has a Hamiltonian cycle. The components of \mathcal{A} are defined as follows: $Q := \bigcup_{i \in [1, n]} \{p_i, q_i, r_i\}$, $\Sigma := \{a_1, a_2, \dots, a_n, \#\}$, $q_0 := p_1$, $F := \bigcup_{i \in [1, n]} \{p_i, q_i\}$, and finally Δ contains the following transitions, for all $i \in [1, n]$:

$$p_i \xrightarrow{a_i} q_i, \quad p_i \xrightarrow{a_j} r_i \text{ for all } j \neq i, \quad q_i \xrightarrow{\#} p_i, \quad \text{and } r_i \xrightarrow{\#} p_k \text{ if } (i, k) \in E .$$

The only non-determinism in \mathcal{A} occurs at the r_i states when reading $\#$: we then have a choice between all the p_k where $(i, k) \in E$.

We give an example for G in Figure 5, where solid lines show the Hamiltonian cycle, together with a construction of \mathcal{A} from G , where solid lines show a determinisation by pruning witnessing this Hamiltonian cycle.

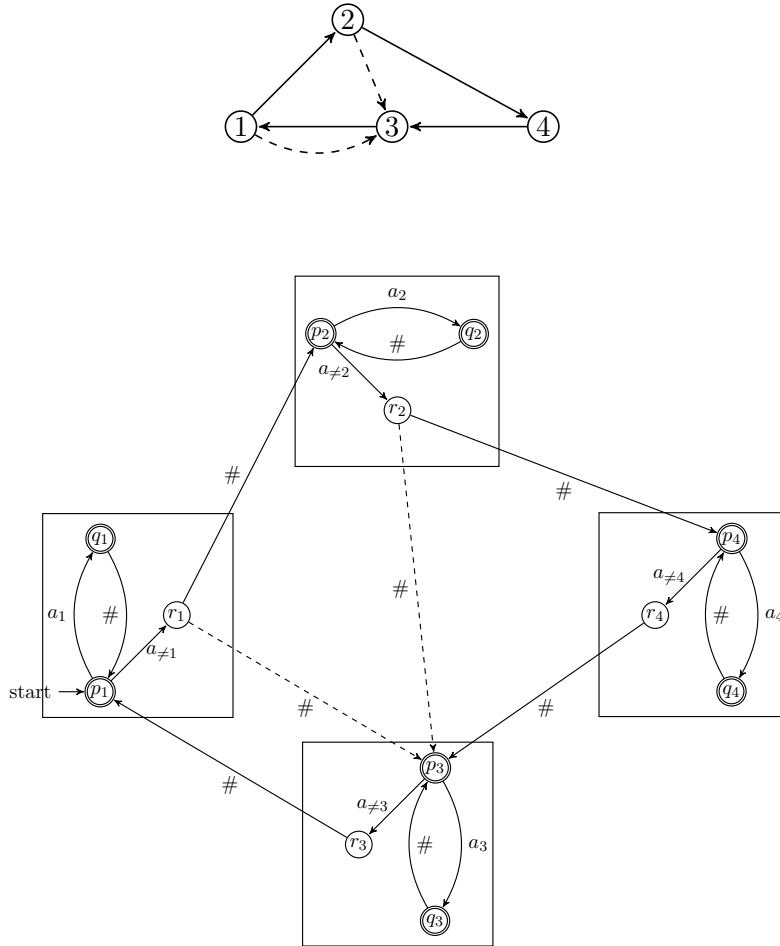


Figure 5: Construction of NCA \mathcal{A} (below) from G (above)

For each $i \in [1, n]$, we can think of the set of states $\{p_i, q_i, r_i\}$ as a cloud in \mathcal{A} representing the vertex i of the graph G .

Let $\Sigma' := \Sigma \setminus \{\#\}$, and $L = \bigcup_{i=1}^n (\Sigma' \#)^* (a_i \#)^\omega$. First note that, provided G is strongly connected, we have $L(\mathcal{A}) = L$. Indeed, for a run to be accepting by \mathcal{A} , it has to visit r_i finitely many times for all i , i.e. after some point it has to loop between p_i and q_i for some fixed i , so the input word must be in L . This shows $L(\mathcal{A}) \subseteq L$. On the other hand, consider a word $w \in L$ of the form $u(a_i \#)^\omega$ with $u \in (\Sigma' \#)^*$. Then \mathcal{A} will have a run on u reaching some cloud j , and since the graph is strongly connected, the run can be extended to the cloud i reading a word of $(a_i \#)^*$. From there, the automaton will read $(a_i \#)^\omega$ while looping between p_i and q_i . We can build an accepting run of \mathcal{A} on any word $w \in L$, so $L \subseteq L(\mathcal{A})$.

Now we shall prove that \mathcal{A} is DBP if and only if G has a Hamiltonian cycle.

(\Rightarrow) Suppose \mathcal{A} is DBP, and let \mathcal{D} be an equivalent DCA obtained from \mathcal{A} by removing transitions. Notice that since the only non-determinism in \mathcal{A} is when reading a $\#$ letter in a r_i state, by construction of \mathcal{A} , building \mathcal{D} corresponds to choosing one out-edge for each vertex of G . This means it induces a set of disjoint cycles in G . We show that it actually is a unique Hamiltonian cycle. Indeed, assume that some vertex of i is not reachable from 1 in G . Equivalently, it means that some cloud i is not reachable from p_1 in \mathcal{D} . This implies that $(a_i\#)^\omega \notin L(\mathcal{D})$, which contradicts $L(\mathcal{D}) = L(\mathcal{A}) = L$. Therefore, \mathcal{D} is strongly connected, and describes a Hamiltonian cycle in G .

(\Leftarrow) Conversely, if G has an Hamiltonian cycle π , we can build the automaton \mathcal{D} accordingly, by setting for all $i \in [1, n]$, $\Delta_{\mathcal{D}}(r_i, \#) = \{p_j\}$ where j is the successor of i in π . Since \mathcal{D} is strongly connected, it still recognises L , and since it is deterministic it is a witness that \mathcal{A} is DBP.

This completes the proof of the fact that \mathcal{A} is DBP if and only if G has a Hamiltonian cycle. Since this is a polynomial time reduction from Hamiltonian Cycle to the DBP property of an NCA, we showed that checking whether an NCA is DBP is NP-hard.

Note that we used $n + 1$ letters here, but it is straightforward to re-encode this reduction using only two letters. Therefore, the problem is NP-hard even on a two-letter alphabet. It is trivially in PTIME on a one-letter alphabet, as there is a unique infinite word. \square

The second part of Theorem 7.8 is given by the following lemma.

Lemma 7.10. *Checking whether an NCA is DBP is in NP.*

Proof. Suppose an NCA \mathcal{A} is given. We want to check whether it is DBP. We do this via the following NP algorithm.

- Nondeterministically prune transitions of \mathcal{A} to get a deterministic automaton \mathcal{D} .
- Check whether $L(\mathcal{A}) \subseteq L(\mathcal{D})$. For that, we check if $L(\mathcal{A}) \cap \overline{L(\mathcal{D})} = \emptyset$

The second step of the algorithm can be done in polynomial time, since it amounts to finding an accepting lasso in $\mathcal{A} \times \overline{\mathcal{D}}$, where $\overline{\mathcal{D}}$ is a Büchi automaton obtained by dualizing the acceptance condition of \mathcal{D} . An accepting lasso is a path of the form $q_0 \xrightarrow{u}^* p \xrightarrow{v}^* p$, witnessing that the word uv^ω is accepted, i.e. in this case the loop should visit only accepting states from the NCA \mathcal{A} , and at least one Büchi state from the DBA $\overline{\mathcal{D}}$. Finding such a lasso is actually in NL. Therefore, the above algorithm is in NP, and its correctness follows from the fact that $L(\mathcal{D}) \subseteq L(\mathcal{A})$ is always true, as any run of \mathcal{D} is in particular a run of \mathcal{A} . \square

8. BÜCHI CASE

Non-deterministic Büchi automata (NBA) correspond to the general case of non-deterministic ω -automata, as they allow to recognise any ω -regular language, and are easily computable from non-deterministic automata with more general acceptance conditions.

We will briefly describe the generalisation of previous constructions here, and explain what is the main open problem remaining to solve in order to obtain a satisfying generalisation. We take Safra's construction [29] as the canonical determinisation for Büchi automata. Safra's construction outputs a Rabin automaton.

The idea behind the previous partial determinisation construction can be naturally adapted to Safra's: it suffices to restrict the image of the Safra tree labellings to sets of states

of size at most k . The bottleneck of the incremental determinisation is then to test whether a Rabin automaton is GFG (or DBP). For DBP checking, the same proof as Theorem 7.8 shows that it is NP-complete. However for GFG checking, the complexity is widely open. It is only currently known to be in P for the particular cases of coBüchi [20] and Büchi [2] conditions. A lower bound for GFG checking with acceptance condition C (for instance C being Parity or Rabin) is the complexity of solving games with winning condition C [20], known to be in QuasiP for parity [7] and NP-complete for Rabin [13]. In both cases, the complexity of solving those games is in P if the acceptance condition C is fixed (for instance $[i, j]$ -parity for fixed i and j). On the other hand, the best known upper bound for the checking the GFG property is EXPTIME [20], even for a fixed acceptance condition C such as Parity with 3 ranks. Finding an efficient algorithm for GFG checking of Rabin (or Parity) automata would therefore be of great interest for this incremental procedure, and would allow to efficiently build GFG automata from NBA.

8.1. Safra Construction. Let \mathcal{A} be an NBA $(Q, \Sigma, \Delta, q_0, F)$ where F is the set of accepting states, and let $n = |Q|$. Safra construction produces an equivalent deterministic Rabin automaton $\mathcal{D} = (Q', \Sigma, \Delta', q'_0, F')$ with $2^{O(n \log n)}$ many states [29, 26].

We recall here the construction, in order to adapt it to an incremental construction computing the width and producing a GFG automaton.

Each state in Q' is a tuple $(T, \sigma, \chi, \lambda)$ where

- $T = (V, v_r, cl)$ is a tree where V is the set of nodes, $v_r \in V$ is the root and cl (for *children list*) is a function $V \rightarrow V^*$ mapping each node to the ordered list of its children, from left to right.
- $\sigma : V \rightarrow 2^Q$ maps each node to a set of states, such that
 - (1) For each $v \in V$, if $cl(v) = v_1 \dots v_k$, then $\sigma(v_1) \cup \dots \cup \sigma(v_k) \subsetneq \sigma(v)$.
 - (2) If v and v' are two nodes such that none of them is ancestor of the other then $\sigma(v)$ and $\sigma(v')$ are disjoint.
 - (3) If $\sigma(v) = \emptyset$, then v must be the root node v_r .

Note that these conditions imply that $|V| \leq n$.

- $\chi : V \rightarrow \{Green, White\}$ assigns a colour to each node.
- $\lambda : V \rightarrow \{l_1, l_2, \dots, l_{2n}\}$ associates a label to each node in V .

The initial state q'_0 is $(T_0, \sigma_0, \chi_0, \lambda_0)$, where T_0 contains only the root v_r , $\sigma_0(v_r) = \{q_0\}$, $\chi_0(v_r) = White$, $\lambda_0(v_r) = l_1$.

Now we define Δ' . The state $(T, \sigma, \chi, \lambda)$, reading $a \in \Sigma$, is moved to a state $(T', \sigma', \chi', \lambda')$ as follows :

- (i) Let $T = (V, v_r, cl)$. First expand the tree T to $T_1 = (V_1, v_r, cl_1)$ as follows: for each node $v \in V$, if $\sigma(v) \cap F \neq \emptyset$, then add a node v' such that v' is the right-most child of v in T_1 .
- (ii) Extend σ to σ_1 as follows: for all $v \in V \cap V_1$, set $\sigma_1(v) = \sigma(v)$. And for each new node $v \in V_1 \setminus V$, let p be the parent node of v in T_1 , set $\sigma_1(v) = \sigma(p) \cap F$.
Extend λ to λ_1 as follows: for all $v \in V \cap V_1$, $\lambda_1(v) = \lambda(v)$. And for each new node $v \in V_1 \setminus V$, fix a new label to v which was not used in V . We can always find such a label since there are $2n$ labels whereas $|V| \leq n$ and each node in V generates atmost one new node in V_1 .
- (iii) For each node $v \in V_1$, apply the subset construction locally, i.e. define $\sigma'_1 : V_1 \rightarrow 2^Q$ such that $\sigma'_1(v) = \Delta(\sigma_1(v), a)$. As before, $\Delta(X, a)$ denotes the set $\{q' \mid \exists q \in X, q' \in \Delta(q, a)\}$.

Now we modify T_1 and σ'_1 so that the structure satisfies the conditions specified for the states of \mathcal{D} as follows:

- (iv) For every node $v \in V$, if there is some $s \in \sigma'_1(v)$ and also $s \in \sigma'_1(v')$ for some node v' such that v and v' have a common ancestor, say u , and v is in the subtree rooted at a child u_1 of u and v' is in the subtree rooted at a child u_2 of u where u_1 is to the left of u_2 , then remove s from $\sigma'_1(v')$. This corresponds to retaining only the “oldest” copy of each active state in the simulation.
- (v) Remove all nodes v such that $\sigma'_1(v) = \emptyset$ and v is not the root.
- (vi) For each node v , if the union of the sets associated with the children of v is equal to $\sigma(v)$, then remove all the children of v and make $\chi(v) = \text{Green}$. And for all other nodes, set $\chi(v) = \text{White}$.

Let the set of remaining nodes be V' . And σ', λ', χ' are retained from the nodes in V' .

If T is a tree, we denote by $V(T)$ its set of nodes.

The Rabin acceptance condition is given by $\{(G_1, B_1), (G_2, B_2), \dots, (G_{2n}, B_{2n})\}$ where G_i are the good states and B_i are the bad states and they are defined as follows:

- $G_i = \{(T, \sigma, \chi, \lambda) \in Q' \mid \exists v \in V(T) : \lambda(v) = l_i \text{ and } \chi(v) = \text{Green}\}$
- $B_i = \{(T, \sigma, \chi, \lambda) \in Q' \mid \forall v \in V(T) : \lambda(v) \neq l_i\}$

That is to say, a run is accepting if there is some i such that states from G_i appear infinitely often while states from B_i appear only finitely often.

The Safra construction is an efficient way to compress information about all possible runs of \mathcal{A} . Indeed, on a word $w = a_0a_1a_2\dots$, the set of runs on w can be described by an infinite Direct Acyclic Graph (DAG) called the run-DAG. This run-DAG has vertices $Q \times \mathbb{N}$, and edges $\{((p, i), (q, i + 1)) \mid (p, a_i, q) \in \Delta\}$. We have that $w \in L(\mathcal{A})$ if and only if there is a path in this run-DAG starting in $(q_0, 0)$ and visiting infinitely many states from F . Safra trees store relevant information about prefixes of this DAG of the form $Q \times [0, i]$, and the acceptance condition of \mathcal{D} is designed to characterize whether the run-DAG of \mathcal{A} contains a Büchi accepting run.

Theorem 8.1. [29] *The deterministic Rabin automaton \mathcal{D} built via the Safra construction is equivalent to \mathcal{A} .*

8.2. Incremental Safra Construction. We can extend the concept of k -subset construction or k -breakpoint construction for NFA and NCA respectively to the Safra construction. We describe below the k -Safra construction, for a parameter $k \leq n$.

Here we restrict $\sigma(v_r)$, v_r being the root, to sets of size at most k . Since all other nodes in Safra trees are labelled by subsets of $\sigma(v_r)$, this implies that the labelling of all the nodes in all Safra trees have size at most k . We define the construction formally as follows:

Given an NBA $\mathcal{A} = (Q, \Sigma, \Delta, q_0, F)$. Define the NRA $\mathcal{A}_k = (Q', \Sigma, \Delta', q'_0, F')$ where each state in Q' is $(T, \sigma, \chi, \lambda)$ such that

- $T = (V, v_r, cl)$ as in the original construction.
- $\sigma : V \rightarrow 2^Q$ satisfying the same properties as before. Additionally we also have the condition that $|\sigma(v)| \leq k$ for all $v \in V$.
- χ and λ are also defined as before.

Now we define Δ' . All the steps remain unchanged except step (iii), which is replaced by the following step, nondeterministically choosing a subset of size k for the root, and propagating it down in the tree:

$$\sigma'_1(v_r) := \begin{cases} \{\Delta(\sigma_1(v_r), a)\} & \text{if } |\Delta(\sigma_1(v_r), a)| \leq k \\ \{X' \mid X' \subseteq \Delta(\sigma_1(v_r), a), |X'| = k\} & \text{otherwise} \end{cases}$$

And for every other node $v \neq v_r$, $\sigma'_1(v) := (\Delta(\sigma_1(v), a)) \cap \sigma'_1(v_r)$.

This corresponds to extending the run-DAG by at most k nodes (the label of the root) at each step, i.e. non-deterministically building a subset of the run-DAG of width at most k .

Initial states and acceptance condition are defined as before.

We can now state that this k -Safra construction characterizes the notion of width in the same way the previous incremental constructions did:

Lemma 8.2. $L(\mathcal{A}) = L(\mathcal{A}_k)$ and $\text{width}(\mathcal{A}) \leq k \iff \mathcal{A}_k$ is GFG.

Proof. Suppose that $w \in L(\mathcal{A})$, witnessed by the run $\rho = q_0q_1q_2 \dots$. Then in \mathcal{A}_k , at step i , choose a set X of size at most k containing q_i as $\sigma(v_r)$ where v_r is the root of the tree and for all other nodes, take its intersection with the label of the root. Since the run-DAG that is built contains a Büchi accepting run, the correctness of the original Safra construction ensures that this run is accepting, so $w \in L(\mathcal{A}_k)$.

Conversely, let $w \in L(\mathcal{A}_k)$. This means that the run-DAG of width k guessed by the automaton contains a Büchi accepting run of \mathcal{A} on w , so $w \in L(\mathcal{A})$.

Now suppose that $\text{width}(\mathcal{A}) \leq k$ and let σ_w be a winning strategy for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$. The only non-determinism in \mathcal{A}_k consists in choosing subsets of size k for the label of the root of Safra trees. So σ_w naturally induces a GFG strategy σ_{GFG} for \mathcal{A}_k , by choosing as root labelling the subset given by σ_w . Again, the correctness of the Safra construction implies that this GFG strategy is correct, as the run-DAG will contain a Büchi accepting run, by definition of the winning condition in $\mathcal{G}_w(\mathcal{A}, k)$.

Conversely, let σ_{GFG} be a GFG strategy for \mathcal{A}_k . It induces a winning strategy σ_w for Player 0 in $\mathcal{G}_w(\mathcal{A}, k)$ which follows the labellings of the root nodes in the run induced by σ_{GFG} . \square

Therefore, we can design a similar incremental approach as in the case of NFA or NCA, and find the minimum k for which the k -Safra construction is GFG, or DBP.

8.3. Complexity of GFG and DBP checking. As mentioned in the beginning of Section 8, the complexity of checking whether an automaton is GFG in the general case of Rabin or Parity automata is widely open. It is only known to be in P for coBüchi condition [20] and for Büchi condition [2], while the upper bound is EXPTIME for higher conditions, such as parity condition with 3 ranks. This means that in the current state of knowledge, the incremental Safra construction combined with GFG checking only yields a doubly exponential upper bound for the width problem of Büchi automata. Recall that the lower bound for the width problem is EXPTIME also in this case, by Remark 6.10. Obtaining a better upper bound via a different route is an interesting problem, that we leave open.

On the other hand, if we aim at obtaining a deterministic automaton, the picture is simplified and matches the coBüchi case. Indeed, for DBP checking, we can generalise Theorem 7.8 stating NP-completeness of DBP checking for coBüchi automata:

Theorem 8.3. *Checking whether a Rabin (or Streett, Muller) automata is DBP is NP-complete.*

Proof. Since coBüchi condition is a particular case of Rabin condition, NP-hardness follows from Lemma 7.9.

We now show membership in NP, in the same way as in Lemma 7.10.

As before, we non-deterministically choose a set of edges to remove in order to obtain a deterministic Rabin automaton \mathcal{D} . Then, it remains to decide emptiness of the automaton $\mathcal{A} \times \overline{\mathcal{D}}$, where \mathcal{A} is a non-deterministic Rabin automaton and \mathcal{D} is a deterministic Streett automaton (Streett is the condition dual to Rabin). Again, emptiness of this automaton amounts to guessing an accepting lasso $q_0 \xrightarrow{u}^* p \xrightarrow{v}^* p$. There is no direct NL algorithm to guess such a lasso as in the proof of Theorem 7.8, since in order to verify the Streett condition of $\overline{\mathcal{D}}$, all Streett pairs must be checked. However, this can be done in NP: guess the lasso, guess the Rabin pair to witness acceptance of \mathcal{A} (or verify them all), and verify that all Streett pairs of $\overline{\mathcal{D}}$ are verified by the loop: for each Streett pair (E, F) , either the loop contains a state from E or no state from F . This procedure can be generalised to Muller acceptance condition for \mathcal{A} : it is always in P to decide whether a particular loop verifies a boolean combination of Muller acceptance conditions. Overall, this yields an NP algorithm for checking whether a Rabin (or Streett, Muller) automaton is DBP. \square

9. CONCLUSION

The width measure can be thought of as a measure of non-determinism in automata, that is essentially orthogonal to ambiguity. It also bounds the number of steps in our incremental determinisation procedures. Therefore, if we know that the width is small, we can obtain a deterministic or GFG automaton without having to go through a full determinization construction as intermediary step. The EXPTIME-completeness of the width problem shows that there is essentially no shortcut that would allow to jump directly to the good level of the incremental construction by computing the optimal width without performing the incremental construction. A dichotomic approach could still present some advantages in practice by saving a few computations in the process of finding the width, but the EXPTIME barrier is a strict theoretical limit.

The cases of finite words and coBüchi condition are especially well-suited for this approach. Indeed, these conditions allow polynomial time checking of the GFG property [24, 20], and polynomial time minimization of the resulting GFG automaton [28]. The NP-completeness of DBP checking for coBüchi automata is another reason to aim for a GFG automaton when determinism is not strictly required.

For Büchi automata, we need to build increasingly complex Rabin automata \mathcal{A}_k via the k -Safra construction. The complexity of checking whether such automata are GFG is less well-understood. As future work, it is pertinent to either search for a special GFG checking procedure well-suited to these \mathcal{A}_k , or on the contrary show that checking the GFG property for these particular \mathcal{A}_k is as hard as for general automata. In any case, this problem provides additional motivation to pinpoint the complexity of GFG checking in general.

Acknowledgements. We thank the anonymous reviewers for their detailed feedback, and Nathalie Bertrand for helpful discussions.

REFERENCES

- [1] Benjamin Aminof, Orna Kupferman, and Robby Lampert. Reasoning about online algorithms with weighted automata. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4-6, 2009*, pages 835–844, 2009.

- [2] Marc Bagnol and Denis Kuperberg. Büchi good-for-games automata are efficiently recognizable. In *Proceedings of the 38th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2018, December 11-13, 2018, Ahmedabad, India*, pages 16:1–16:14, 2018.
- [3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.
- [4] Udi Boker, Denis Kuperberg, Orna Kupferman, and Michał Skrzypczak. Nondeterminism in the presence of a diverse or unknown future. In *Proceedings of the 40th International Colloquium on Automata, Languages, and Programming, ICALP 2013, Riga, Latvia, July 8-12, 2013, Part II*, pages 89–100, 2013.
- [5] Udi Boker, Orna Kupferman, and Michał Skrzypczak. How deterministic are good-for-games automata? In *Proceedings of the 37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*, pages 18:1–18:14, 2017.
- [6] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 457–468, 2013.
- [7] Cristian S. Calude, Sanjay Jain, Bakhadyr Khossainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 252–263, 2017.
- [8] Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesis. *Journal of Symbolic Logic*, 28(4):289–290, 1963.
- [9] Lorenzo Clemente. *Generalized simulation relations with applications in automata theory*. PhD thesis, University of Edinburgh, UK, 2012.
- [10] Thomas Colcombet. The theory of stabilisation monoids and regular cost functions. In *Proceedings of the 36th International Colloquium on Automata, Languages, and Programming, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Part II*, pages 139–150, 2009.
- [11] Thomas Colcombet. Forms of determinism for automata (invited talk). In *Proceedings of the 29th International Symposium on Theoretical Aspects of Computer Science, STACS 2012, February 29th - March 3rd, 2012, Paris, France*, pages 1–23, 2012.
- [12] Thomas Colcombet and Nathanaël Fijalkow. Universal graphs and good for games automata: New tools for infinite duration games. In *Proceedings of the 22nd International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019*, pages 1–26, 2019.
- [13] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs (extended abstract). In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 328–337, 1988.
- [14] Kousha Etessami. A hierarchy of polynomial-time computable simulations for automata. In *Proceedings of the 13th International Conference, CONCUR 2002 - Concurrency Theory, Brno, Czech Republic, August 20-23, 2002*, pages 131–144, 2002.
- [15] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.
- [16] Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In *Proceedings of the 20th International Workshop on Computer Science Logic, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, pages 395–410, 2006.
- [17] Simon Iosti and Denis Kuperberg. Eventually safe languages. In *Developments in Language Theory, 23rd International Conference, DLT 2019*, LIPIcs, page to appear. Springer International Publishing, 2019.
- [18] Joachim Klein, David Müller, Christel Baier, and Sascha Klüppelholz. Are good-for-games automata good for probabilistic model checking? In *Proceedings of the 8th International Conference on Language and Automata Theory and Applications, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings*, pages 453–465, 2014.
- [19] Denis Kuperberg and Anirban Majumdar. Width of non-deterministic automata. In *Proceedings of the 35th International Symposium on Theoretical Aspects of Computer Science, STACS 2018, February 29th - March 3rd, 2018, Caen, France*, 2018.

- [20] Denis Kuperberg and Michał Skrzypczak. On determinisation of good-for-games automata. In *Proceedings of the 42nd International Colloquium on Automata, Languages, and Programming, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 299–310, 2015.
- [21] Hing Leung. Separating exponentially ambiguous finite automata from polynomially ambiguous finite automata. *SIAM Journal of Computing*, 27(4):1073–1082, 1998.
- [22] Hing Leung. Descriptive complexity of nfa of different ambiguity. *International Journal of Foundations of Computer Science*, 16(05):975–984, 2005.
- [23] Hing Leung. Structurally unambiguous finite automata. *Lecture Notes in Computer Science*, 4094 LNCS:198–207, 2006.
- [24] Christof Löding and Stefan Repke. Decidability results on the existence of lookahead delegators for NFA. In *Proceedings of the 33th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*, pages 327–338, 2013.
- [25] Satoru Miyano and Takeshi Hayashi. Alternating finite automata on ω -words. *Theoretical Computer Science*, 32(3):321–330, 1984.
- [26] Madhavan Mukund. Finite-state automata on infinite inputs. In *Modern Applications of Automata Theory*, pages 45–78. 2012.
- [27] Michael Oser Rabin. *Automata on Infinite Objects and Church’s Problem*. American Mathematical Society, Boston, MA, USA, 1972.
- [28] Bader Abu Radi and Orna Kupferman. Minimizing gfg transition-based automata. In *Proceedings of the 46th International Colloquium on Automata, Languages, and Programming, ICALP 2019*, LIPIcs, page to appear. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.
- [29] Shmuel Safra. On the complexity of omega-automata. In *Proceedings of the 29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 319–327, 1988.
- [30] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal of Computing*, 8(2):151–174, 1979.
- [31] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theoretical Computer Science*, 88(2):325 – 349, 1991.