Formal Analysis of Online Algorithms[⋆]

Benjamin Aminof[1], Orna Kupferman[1], and Robby Lampert[2]

[1] School of Computer Science and Engineering,Hebrew University, Jerusalem 91904, Israel
[2] Department of Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel

Abstract. In [AKL10], we showed how viewing online algorithms as reactive systems enables the application of ideas from formal verification to the competitive analysis of online algorithms. Our approach is based on weighted automata, which assign to each input word a cost in $\mathbb{R}^{\geq 0}$. By relating the "unbounded look ahead" of optimal offline algorithms with nondeterminism, and relating the "no look ahead" of online algorithms with determinism, we were able to solve problems about the competitive ratio of online algorithms and the memory they require.

In this paper we improve the application in three important and technically challenging aspects. First, we allow the competitive analysis to take into account assumptions about the environment. Second, we allow the online algorithm to have a bounded lookahead. Third, we describe a symbolic version of the model-checking algorithm and demonstrate its applicability. The first two contributions broaden the scope of our approach to settings in which the traditional analysis of online algorithms is particularly complicated. The third contribution improves the practicality of our approach and enables it to handle larger state spaces.

1 Introduction

In *formal verification*, we verify that a system has a desired property by checking whether a model of the system satisfies a formal specification of the property. An important feature of formal verification is that it enables reasoning about *reactive systems*, which maintain an on-going interaction with their environment [HP85].

Online algorithms for optimization problems can be viewed as reactive systems. An online algorithm processes requests in real-time: At each round, the environment issues a request, and the algorithm should process it. The sequence of requests is not known in advance, and the goal of the algorithm is to minimize the overall cost of processing all the requests in the sequence. For example, in the *paging* problem, we have a two-level memory hierarchy: A slow memory that contains n different pages, and a *cache* that contains at most k different pages (typically, $k \ll n$). Pages that are in the cache can be accessed at zero cost. If a request is made to access a page that is not in the cache, the page should be brought into the cache, at a cost of 1, and if the cache is full, some other page should first be evicted from the cache. The paging problem is, given a sequence of requested pages, to decide which page to evict whenever an eviction is needed. The goal is to minimize the total cost. Online algorithms for many problems have already been extensively studied for several decades, and have aroused much interest, both from a practical and a theoretical point of view [BEY98].

The interaction described above between an online algorithm and its environment is at the heart of formal verification. Still, the questions that are traditionally answered by formal-verification techniques are very different from those that are asked in the context of online algorithms. In formal verification, a system is checked with respect to a given specification. The specification can be qualitative (e.g., "whenever a request to a page is made, and this page is not in the cache, the page is brought into the cache") or quantitative (e.g., "what is the maximal number of page faults within a window of k rounds?") [CCH$^+$05]. The most interesting question about an online algorithm, however, is of a different nature, and refers to its *competitive ratio*: the worst-case (with respect to all input sequences) ratio between the cost of the algorithm and the cost of an optimal solution (one that may be given by an *offline algorithm*, which knows the input sequence in advance). Thus, we can specify the model-checking

[⋆] Supported in part by the Minerva Center at the Weizmann Institute, and by an Advanced Research Grant from the ERC under the European Community's 7th Framework Programme.

problem of online algorithms as follows: Consider an optimization problem P. Given an algorithm g and a competitive ratio α, is g α-competitive with respect to an optimal offline algorithm for P?

Recently, we extended the scope of formal verification to reasoning about online algorithms [AKL10]. The approach in [AKL10] is based on *weighted finite automata* (WFAs, for short) [KS86,Moh97]. A WFA $\mathcal{A}$ induces a partial *cost* function from Σ^* to $\mathbb{R}^{\geq 0}$. Technically, each transition of $\mathcal{A}$ has a cost associated with it. The cost of a run is the sum of the costs of the transitions taken along the run, and the cost of a word w, denoted $cost(\mathcal{A}, w)$, is the minimum cost over all accepting runs on it (the cost is undefined if no run on the word is accepting). Consider an optimization problem P with requests in Σ. An algorithm for P can be viewed as a mapping of words in Σ^+ to a set of actions available to the algorithm [BDBK$^+$94]. For a finite set S of configurations, we say that an algorithm uses memory S if there is a regular mapping of Σ^* into S such that the algorithm behaves in the same manner on identical continuations of words that are mapped to the same configuration.

The set of online algorithms for P that use memory S induces a WFA $\mathcal{A}_P$, with alphabet Σ and state space S, such that the transitions of $\mathcal{A}_P$ correspond to actions of the algorithms and the cost of each transition is the cost of the corresponding action. It is shown in [AKL10] that many optimization problems have algorithms that use finite memory and can be modeled by weighted automata as described above. Moreover, the "unbounded look ahead" of the optimal offline algorithm corresponds to nondeterminism in $\mathcal{A}_P$, and the "no look ahead" of online algorithms corresponds to deterministic automata embeded-ded in $\mathcal{A}_P$. Consequently, questions about the competitive ratio of online algorithms can be reduced to questions about *determinization* and *approximated determinization* of WFAs [AKL11]. In particular, the model-checking problem for an online algorithm g can be reduced to the problem of deciding whether the pruning of $\mathcal{A}_P$ induced by g results in a deterministic automaton $\mathcal{A}_P^g$ that α-approximates $\mathcal{A}_P$ (that is, the automaton $\mathcal{A}_P^g$ accepts the same set of words as $\mathcal{A}_P$, and $cost(\mathcal{A}_P^g, w) \leq \alpha \cdot cost(\mathcal{A}_P, w)$ for all words w in this set). In addition, the synthesis problem for online algorithms can be reduced to the problem of deciding whether $\mathcal{A}_P$ contains an embedded deterministic automaton that α-approximates $\mathcal{A}_P$.

The competitive analysis of online algorithms takes into account the most hostile environment. Indeed, an online algorithm g is α-competitive if its cost with respect to every input sequences is at most α times the cost of an optimal solution. Quite often, however, the nature of the problem restricts the set of possible input sequences. Much research has been carried out in the online-algorithm community studying the competitive analysis of online algorithms under different assumptions about the environment [BEY98]. For example, for the paging problem, Borodin et al. studied the *access graph model* [BIRS95], which takes into account the *locality of reference* principle. In the access graph model, the paging problem is equipped with a graph whose vertices are the pages, and two pages can be requested successively only if they are connected in the graph.

The first contribution of this paper is an extension of the framework in [AKL10] to a setting in which assumptions about the environment can be taken into account. The issue of restricted environments is well studied in formal verification. Ideas like fairness [Fra86], assume-guarantee reasoning [Pnu85], and synthesis under restricted environments [CHJ08], have been suggested in order to take assumptions about the environment into account. We study the competitive analysis of online algorithms in which assumptions about the environment are given by means of a *nondeterministic finite automaton* (NFA, for short). In this setting, the competitive ratio of an online algorithm is defined only with respect to input sequences that belong to the language of the assumption NFA. Our definition generalizes restrictions such as the one induced by the access graph — it supports all regular assumptions. In addition, it nicely combines with the automata-based approach initiated in [AKL10]. Consider an online problem P, a set of configurations S for it, an approximation factor α, an online algorithm g that uses configurations in S, and an assumption NFA $\mathcal{U}$. We show that the problem of deciding whether g is α-competitive with respect to input sequences in $L(\mathcal{U})$ (*model checking with assumptions*) can be solved in polynomial time. On the other hand, the problem of deciding whether there is an online algorithm that uses configurations in S and is α-competitive with respect to input sequences in $L(\mathcal{U})$ (*synthesis with assumptions*) is NP-complete. We note that NP-hardness holds already for unweighted automata and $\alpha = 1$, and even when $\mathcal{U}$ is deterministic. This is in contrast to the setting with no assumptions studied in [AKL10], in which synthesis with $\alpha = 1$ can be solved in polynomial time. Thus, interestingly, the addition of assumptions makes the problem substantially more complex.

The second contribution of this paper is an extension of the framework in [AKL10] to a setting in which the online algorithm has a *bounded lookahead* on the requests yet to come. Since an offline algorithm can be viewed as an online algorithm with an unbounded lookahead, the setting of a bounded lookahead covers the "middle-ground" between onlineness and offlineness. However, considering online algorithms with lookahead is also interesting from a practical point of view. In practical applications, requests do not always arrive one by one, but sometimes naturally occur in bursts. Also, some applications benefit from delaying requests so that a block of requests can be served all at once, minimizing common overhead. Finally, in some applications requests are generated faster than they can be served, and thus the online algorithm has to maintain a buffer containing requests that are pending service. The challenges of manually analyzing online algorithms are even bigger in the setting of lookahead [Alb97,Bre98,You91]. Indeed, the analysis has to take into an account the extended memory of the algorithm and the partition of the input stream to requests that are in the lookahead and those that are not. The automata-theoretic approach can be naturally extended to handle bounded lookahead in online algorithms by means of automata with a bounded lookahead. Such automata read, in each transition, a sequence of the next $l + 1$ letters, for a fixed parameter l (that is, the look ahead). We study the problems of determinization and approximated determinization of nondeterministic weighted automata with a bounded lookahead, and how questions about online algorithms can be reduced to them. Unfortunately, the analysis is exponential in the lookahead. A similar computational cost is needed in the analysis of lookahead in regular infinite games [HKT10], and we prove that the cost indeed cannot be polynomial.

One of the main challenges in formal verification is the need to cope with very big, often infinite, state spaces. In our context, the state space often involves weights, and is thus very big. The third contribution of the paper is a description of a *symbolic algorithm* [BCM$^+$92] for the problem of model-checking of online algorithms. In symbolic reasoning, the state space and the transitions of the system are given symbolically by characteristic functions over a set of variables that encode the state space of the system. The operations allowed to the verification algorithm correspond to manipulations of predicates over the set of variables. The fact a symbolic algorithm has to manipulate predicates over variables forces it to refer to sets of elements rather than to individual elements. The idea behind the algorithm is as follows. Consider a WFA $\mathcal{A}$. We say that a state q of $\mathcal{A}$, (α, i, t)-approximates a state q', for a competitive ratio α, an integer $i \geq 0$, and an additive factor t, if there is a deterministic automaton $\mathcal{A}^q$ with initial state q that is embedded in $\mathcal{A}$ and in which $cost(\mathcal{A}^q, w) \leq t + \alpha \cdot cost(\mathcal{A}^{q'}, w)$ for every word w of length at most i, where $\mathcal{A}^{q'}$ is $\mathcal{A}$ with initial state q'. We show that given a symbolic representation of pairs $\langle q, q' \rangle$ such that q (α, i, t)-approximates q', it is possible to generate a symbolic representation of pairs $\langle q, q' \rangle$ such that q $(\alpha, i+1, t')$-approximates q', for the minimal t' for which such an approximation exists. Note that $t' \geq t$. The symbolic algorithm then calculates a fixed-point of the above transformation. In the process, it detects cycles along which $\mathcal{A}^{q'}$ is "unboundedly better" than $\mathcal{A}^q$. The algorithm then concludes that t' should be increased to infinity. Finally, the answer to the model-checking problem is positive iff there is an initial state q such that q $(\alpha, i, 0)$-approximates q' for all the initial states q' of $\mathcal{A}$ and the iteration i in which a fixed-point was reached[3]. The symbolic implementation can handle also assumptions about the environment and algorithms with lookahead. We implemented our symbolic algorithm, and describe its application in reasoning about two online algorithms for the paging problem.

[3] In [CL92], the authors use an iterative (non-symbolic) procedure that checks for α-competitive algorithms to the server problem. There, a fixed-point has been reached iff such an algorithm exists. By [AKL10], the procedure can be terminated after two rounds of quadratically many iterations.

2 Preliminaries

2.1 Weighted Automata

Standard automata map words in Σ^* to either "accept" or "reject". A weighted automaton can be viewed as a partial function (defined only for accepted words) from Σ^* to $\mathbb{R}^{\geq 0}$. Formally, a *weighted finite automaton* (WFA, for short) is a 6-tuple $\mathcal{A} = \langle \Sigma, Q, \Delta, c, Q_0, F \rangle$, where Σ is a finite input alphabet, Q is a finite set of states, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $c : \Delta \to \mathbb{R}^{\geq 0}$ is a cost function, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. A transition $d = \langle q, a, p \rangle \in \Delta$ (also written as $\Delta(q, a, p)$) can be taken when $\mathcal{A}$ reads the input letter a, and it causes $\mathcal{A}$ to move from state q to state p with *cost* $c(d)$. The transition relation Δ induces a transition function $\delta : Q \times \Sigma \to 2^Q$ in the expected way. Thus, for a state $q \in Q$ and a letter $a \in \Sigma$, we have $\delta(q, a) := \{p : \Delta(q, a, p)\}$. A WFA $\mathcal{A}$ may be nondeterministic in the sense that it may have many initial states, and that for some $q \in Q$ and $a \in \Sigma$, it may have $\Delta(q, a, p_1)$ and $\Delta(q, a, p_2)$, with $p_1 \neq p_2$. If $|Q_0| = 1$ and for every state $q \in Q$ and letter $a \in \Sigma$ we have $|\delta(q, a)| \leq 1$, then $\mathcal{A}$ is a *deterministic* weighted finite automaton (DWFA, for short).

For a word $w = w_1 \ldots w_n \in \Sigma^*$, a run of $\mathcal{A}$ on w is a sequence $r = r_0 r_1 \ldots r_n \in Q^+$, where $r_0 \in Q_0$ and for every $1 \leq i \leq n$, we have $\langle r_{i-1}, w_i, r_i \rangle \in \Delta$. The run r is accepting if $r_n \in F$. The word w is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The (unweighted) *language* of $\mathcal{A}$ is $L(\mathcal{A}) = \{w : w \text{ is accepted by } \mathcal{A}\}$. The cost of an accepting run is the sum of the weights of the transitions that constitute the run. Formally, let $r = r_0 r_1 \ldots r_n$ be an accepting run of $\mathcal{A}$ on w, and let $d = d_1 \ldots d_n \in \Delta^*$ be the corresponding sequence of transitions. The cost of r is $cost(\mathcal{A}, r) = \sum_{i=1}^{n} c(d_i)$. The cost of w, denoted $cost(\mathcal{A}, w)$, is the minimal cost over all accepting runs of $\mathcal{A}$ on w. Thus, $cost(\mathcal{A}, w) = \min\{cost(\mathcal{A}, r) : r \text{ is an accepting run of } \mathcal{A} \text{ on } w\}$.

For two WFAs $\mathcal{A}_1 = \langle \Sigma, Q_1, \Delta_1, c_1, Q_1^0, F_1 \rangle$ and $\mathcal{A}_2 = \langle \Sigma, Q_2, \Delta_2, c_2, Q_2^0, F_2 \rangle$, and $\alpha \geq 1$, we say that $\mathcal{A}_2$ *α-approximates* $\mathcal{A}_1$ if $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ and for all words w in both languages, we have $cost(\mathcal{A}_2, w) \leq \alpha \cdot cost(\mathcal{A}_1, w)$. We say that $\mathcal{A}_2$ is *embedded* in $\mathcal{A}_1$ if $Q_2 = Q_1$, $Q_2^0 \subseteq Q_1^0$, $\Delta_2 \subseteq \Delta_1$, c_2 agrees with c_1 on Δ_2, and $F_1 = F_2$. Thus, $\mathcal{A}_2$ can be obtained from $\mathcal{A}_1$ by decreasing its nondeterminism. Finally, given an approximation factor $\alpha \geq 1$, we say that $\mathcal{A}$ is *α-determinizable by pruning* (α-DBP, for short) if $\mathcal{A}$ has an embedded DWFA that α-approximates $\mathcal{A}$.

2.2 Online Algorithms

A *problem* associates with each possible input I a set $F(I)$ of feasible solutions. In an *optimization problem* (of cost minimization), each solution in $F(I)$ has a cost in $\mathbb{R}^{\geq 0}$, and the goal is to find a feasible solution that minimizes the cost.

An *online algorithm* for an optimization problem P is an algorithm that gets as input a finite sequence of requests, and has to process each request (and end up in a feasible solution) without knowing the requests yet to come. In contrast, an *offline algorithm* for P gets the entire sequence in advance, and its decisions as to how to process a request may depend on the requests yet to come.

Formally, if we denote by Σ the set of requests, and denote by Γ the set of actions that are available to the algorithm, then an online algorithm corresponds to a function $g : \Sigma^+ \to \Gamma$. The processing of an input sequence $\sigma_1 \ldots \sigma_n$ by g is then $g(\sigma_1)$, $g(\sigma_1\sigma_2)$, $g(\sigma_1\sigma_2\sigma_3), \ldots$. In typical optimization problems, there is a cost function $action\_cost : \Gamma \to \mathbb{R}^{\geq 0}$ that associates a cost with each action. The cost of processing an input sequence is the sum of the costs of the actions taken in order to process it. The performance of an online algorithm is typically worse than that of an offline algorithm for the same problem. For analyzing the performance of online algorithms we use *competitive analysis,* which compares the two performance values.

For an online algorithm g and an input $w \in \Sigma^+$, let $g(w)$ denote the cost of processing w by g, and let $\mathrm{OPT}(w)$ denote the cost of processing w by the optimal offline algorithm. We say that an online algorithm g is *α-competitive* if there exists a constant β such that for all input sequences $w \in \Sigma^+$

we have that $g(w) \leq \alpha \cdot \mathrm{OPT}(w) + \beta$. The *competitive ratio* of g is the smallest α for which g is α-competitive. In the rest of the paper we restrict attention to the multiplicative factor α and ignore the additive factor β, except for places where it is not immediately clear how to handle β.

2.3 An Automata-Theoretic Approach to Reasoning about Online Algorithms

Recall that an online algorithm corresponds to a function $g : \Sigma^+ \to \Gamma$ that maps sequences of requests (the history of the interaction so far) to an action to be taken. For a finite set S of configurations, we say that g *uses memory* S, if there is a regular mapping of Σ^* into S such that g behaves in the same manner on identical continuations of words that are mapped to the same configuration.We model the set of online algorithms that use memory S and solve an optimization problem P with requests in Σ and actions in Γ, by a WFA $\mathcal{A}_P = \langle \Sigma, S, \Delta, c, S_0, S \rangle$, where Δ and c describe transitions between configurations and their costs, and S_0 is a set of possible initial configurations. Formally, $\Delta(s, \sigma, s')$ if the set $\Gamma' \subseteq \Gamma$ of actions that process the request σ from configuration s by updating the configuration to s' is non-empty, in which case $c(\langle s, \sigma, s' \rangle) = \min_{\gamma \in \Gamma'} \mathit{action\_cost}(\gamma)$. Note that all the states of $\mathcal{A}_P$ are accepting. Thus, $\mathcal{A}_P$ assigns a cost to all sequences in Σ^*.

As demonstrated in [AKL10], many optimization problems have online algorithms that require finite memory. Below we describe the modeling of the paging problem, presented in Section 1.

Example 1 [**The paging problem** [ST85]] A paging problem P with parameters n (number of pages) and k (size of the cache) induces a WFA $\mathcal{A}_P = \langle \Sigma, S, \Delta, c, S_0, S \rangle$, where $\Sigma = \{1, \ldots, n\}$ is the set of possible requests (page indices), $S = \{C \subseteq \{1, \ldots, n\} : |C| \leq k\}$ is a set of finite configurations, each describing the set of pages currently in the cache, Δ and c describe how (and at which cost) requests are served, and $S_0 = \{\emptyset\}$, indicating that the cache is initially empty. Thus, $\Delta(C, i, C')$ iff one of the following holds: (1) $i \in C$, in which case $C' = C$ and $c(\langle C, i, C' \rangle) = 0$, (2) $i \notin C$, $|C| < k$, and $C' = C \cup \{i\}$, in which case $c(\langle C, i, C' \rangle) = 1$, or (3) $i \notin C$, $|C| = k$, and there is $j \in C$ such that $C' = (C \setminus \{j\}) \cup \{i\}$, in which case $c(\langle C, i, C' \rangle) = 1$. Note that by the definition of S, a configuration stores only the set of pages currently in the cache, and there are no provisions for storing any extra information such as time-stamps, etc. A different automaton for the problem could have defined S in a way that allows the storage of such extra information. We will elaborate on this point in the sequel.

Note that the above modeling restricts attention to *lazy* (a.k.a. demand paging) algorithms, which minimize the change of configurations so that only the current request is served. By [MMS90], for every non-lazy algorithm, there exists a lazy one that performs at least as well.

Let P be an optimization problem, and let $\mathcal{A}_P = \langle \Sigma, S, \Delta, c, S_0, S \rangle$ be a WFA for its algorithms that use memory S. Given a finite sequence of requests $w \in \Sigma^*$, each run of $\mathcal{A}_P$ on w corresponds to a way of serving the requests in w by an algorithm with configurations in S. The set of all runs includes all such algorithms, thus the cost of w in $\mathcal{A}_P$ is the cost of w in an optimal offline algorithm whose configurations are based on S (the configurations of the offline algorithm may also maintain the suffix of the input yet to be processed. This information, however, would be implicit in the nondeterminism of $\mathcal{A}_P$). On the other hand, an online algorithm has to process each request as soon as it arrives, without knowing the requests yet to arrive. Accordingly, an online algorithm that uses memory S corresponds to a DWFA embedded in $\mathcal{A}_P$ (note that this correspondence is lost if we consider unrestricted determinization of $\mathcal{A}_P$). Formally, given an online algorithm $g : \Sigma^+ \to \Gamma$ that uses memory S, let $h : \Sigma^* \to S$ be the regular mapping that witnesses that g uses memory S. Then, the DWFA embedded in $\mathcal{A}_P$ and induced by g is an automaton $\mathcal{A}_P^g$ in which, for all states $s \in S$ and requests $\sigma \in \Sigma$, we have $\delta(s, \sigma) = s'$, where s' is the configuration obtained by applying the action $g(w \cdot \sigma)$ from s, and w is such that $h(w) = s$. In other words, for all $w \in \Sigma^*$, we have $\delta(h(w), \sigma) = h(w \cdot \sigma)$.

Theorem 2. [AKL10] *Given an online problem P and a set S of configurations, let $\mathcal{A}_P$ be a WFA, with state space S, that models online algorithms for P that use memory S. An online algorithm g, that uses memory S, is α-competitive iff $\mathcal{A}_P^g$ α-approximates $\mathcal{A}_P$.*

Note that the setting describes above forces the online algorithm to have the same state space as the offline one. In [AKL10] we described how the framework can handle also online algorithms with a richer state space. The same idea can be applied to the extensions studied in the current paper.

3 Adding Assumptions on the Environment

As discussed in Section 1, an online algorithm can be viewed as a reactive system. The fact that a reactive system has to satisfy its specification with respect to all input sequences is analogous to the fact that an α-competitive online algorithm has to satisfy $g(w) \leq \alpha \cdot \text{OPT}(w)$ for all input sequences $w \in \Sigma^+$. When reasoning about reactive systems, it is sometimes desirable to restrict the universal quantification over all input sequences to a subset of the possible inputs. The automata-theoretic approach naturally formalizes such assumptions in the context of online algorithms. We begin our study with unweighted automata, where things are typically simpler, and then move to weighted automata, which immediately translates to the context of online algorithms.

Given two NFAs, $\mathcal{A}$ and $\mathcal{U}$, we say that $\mathcal{A}$ is *determinizable by pruning with respect to assumptions in* $\mathcal{U}$ ($\mathcal{U}$-DBP, for short), if $\mathcal{A}$ has an embedded DFA $\mathcal{A}'$ such that $L(\mathcal{A}) \cap L(\mathcal{U}) \subseteq L(\mathcal{A}')$. Thus, $\mathcal{A}$ is $\mathcal{U}$-DBP if it can be pruned to a deterministic automaton that accepts all the words in $L(\mathcal{A})$ that are also in $L(\mathcal{U})$. In this case we say that $\mathcal{A}'$ is a *witness* for $\mathcal{A}$ being $\mathcal{U}$-DBP. Similarly, for the weighted case, given a WFA $\mathcal{A}$, an NFA $\mathcal{U}$, and an approximation factor $\alpha \geq 1$, we say that $\mathcal{A}$ is α-$\mathcal{U}$-DBP if $\mathcal{A}$ has an embedded DWFA $\mathcal{A}'$ such that for all $w \in L(\mathcal{A}) \cap L(\mathcal{U})$ we have $cost(\mathcal{A}', w) \leq \alpha \cdot cost(\mathcal{A}, w)$. Intuitively, the NFA $\mathcal{U}$ specifies assumptions about the environment. In particular, usual determinization by pruning is a special case of the above, with $L(\mathcal{U}) = \Sigma^*$.

The *relaxed-α-DBP* problem is to decide, given a WFA $\mathcal{A}$, an approximation factor $\alpha \geq 1$, and an NFA $\mathcal{U}$, whether $\mathcal{A}$ is α-$\mathcal{U}$-DBP. The *relaxed-α-DBP witness-checking* problem is to decide, given a WFA $\mathcal{A}$, an NFA $\mathcal{U}$, $\alpha \geq 1$, and a DFA (DWFA) $\mathcal{A}'$ embedded in $\mathcal{A}$, whether $\mathcal{A}'$ is a witness for $\mathcal{A}$ being α-$\mathcal{U}$-DBP. When $\mathcal{A}$ and $\mathcal{A}'$ are NFAs (that is, unweighted), no approximation factor is given and we refer to the problems as the *relaxed-DBP* and the *relaxed-DBP witness-checking* problems. The relaxed-α-DBP problem corresponds to the synthesis problem, whereas the witness-checking problem corresponds to model checking. We start with the relaxed-DBP and the relaxed-α-DBP witness-checking problems:

Theorem 3. *The relaxed-DBP (relaxed-α-DBP) witness-checking problem is NLOGSPACE-complete (in* PTIME*, respectively).*

Proof: We first prove that the relaxed-DBP witness-checking problem is NLOGSPACE-complete, and we start with the upper bound. Given two NFAs $\mathcal{A}$ and $\mathcal{U}$, and a DFA $\mathcal{A}'$ embedded in $\mathcal{A}$, we have that $\mathcal{A}'$ is a witness for $\mathcal{A}$ being $\mathcal{U}$-DBP iff $L(\mathcal{A}) \cap L(\mathcal{U}) \subseteq L(\mathcal{A}')$. Since $\mathcal{A}'$ is deterministic, its complementation is immediate, and thus, checking the above can be done in NLOGSPACE.

In order to prove NLOGSPACE-hardness, we describe a reduction from the non-reachability problem (proved to be NLOGSPACE-hard in [Imm88]) to the DBP witness-checking problem. Consider a directed graph $G = \langle V, E \rangle$ and two vertices $s, t \in V$. Let $\mathcal{B}$ be some fixed NFA that has a single initial state and is not DBP, and let σ be a letter not in the alphabet of $\mathcal{B}$. Consider the NFA $\mathcal{B}'$ obtained form G by labeling all its edges by σ, adding a self loop (labeled by σ) to s, defining s to be the only initial state, defining all the vertices of G as accepting states, and "plugging $\mathcal{B}$ in t" (that is, adding to t the transitions that exit the initial state of $\mathcal{B}$). Note that if t is not reachable from s, then the language of $\mathcal{B}'$ is σ^*. If t is reachable from s, then the language of $\mathcal{B}'$ is $\sigma^* + \sigma^* \cdot L(\mathcal{B})$. Since $\mathcal{B}$ is not DBP, it is not hard to see that t is not reachable from s iff $\mathcal{B}'$ is DBP.

We now move to the weighted case and prove that the relaxed-α-DBP witness-checking problem is in PTIME. We describe a polynomial algorithm for solving the problem. Given a WFA $\mathcal{A}$, an NFA $\mathcal{U}$, $\alpha \geq 1$, and a DWFA $\mathcal{A}'$ embedded in $\mathcal{A}$, we first construct the WFA $\mathcal{A} \times \mathcal{U}$ that accepts the language $L(\mathcal{A}) \cap L(\mathcal{U})$ with the weights of $\mathcal{A}$. Thus, the states and transitions are defined as in the classical product automaton, and the weights are induced from those in $\mathcal{A}$. Formally, let Q_1 and Q_2 be the state spaces of $\mathcal{A}$ and $\mathcal{U}$, respectively. Then, for $q_1, q_1' \in Q_1, q_2, q_2' \in Q_2$, and $\sigma \in \Sigma$, we set the cost of the transition $(\langle q_1, q_2 \rangle, \sigma, \langle q_1', q_2' \rangle)$ to be the cost of the transition (q_1, σ, q_1') in $\mathcal{A}$. Since the automaton $\mathcal{A}'$ is embedded also in $\mathcal{A} \times \mathcal{U}$, the relaxed-$\alpha$-DBP witness-checking problem reduces to the α-DBP witness-checking problem with respect to $\mathcal{A}'$ and $\mathcal{A} \times \mathcal{U}$. By [AKL10], the latter can be solved in polynomial time. $\square$

We now proceed to the relaxed-DBP problem. In the setting with no assumptions about the environment, it was shown in [AKL10] that the DBP-problem is polynomial for the unweighted case or for the weighted case with $\alpha = 1$, and is NP-complete for the weighted case with $\alpha > 1$. As the following theorem shows, adding assumptions makes the relaxed-DBP problem NP-complete already for the unweighted case, and thus significantly harder. On the positive side, adding assumptions does not make the problem harder in the weighted case with $\alpha > 1$, where it stays NP-complete, as in the setting with no assumptions.

Theorem 4. *The relaxed-DBP and the relaxed-α-DBP problems are NP-complete.*

Proof: First, observe that the problems are in NP since given $\mathcal{A}$ and $\mathcal{U}$, we can guess a DFA $\mathcal{A}'$ embedded in $\mathcal{A}$ and check whether it is a witness. By the above, this can be done in polynomial time.

In order to show that the problems are NP-hard, we describe a reduction from 3SAT to the relaxed-DBP problem. Let θ be a 3CNF formula with m clauses, $c_1, \ldots, c_m$, over the variables $x_1, \ldots, x_n$. We construct an NFA $\mathcal{A}^\theta$ and a DFA $\mathcal{U}^\theta$ over the alphabet $\{\#, 1, ..., m\}$, such that $\mathcal{A}^\theta$ is $\mathcal{U}^\theta$-DBP iff θ is satisfiable.

The NFA $\mathcal{A}^\theta$ has the form of a DAG with four levels (see Figure 1 for an example). On the first level of the DAG there is a single initial state q_0. On the second level there are n states, $x_1, \ldots, x_n$, corresponding to the variables in θ. For each state x_i, there are m transitions, labeled $1, \ldots, m$ from q_0 to x_i. On the third level there are $2n$ states, $1_{true}, 1_{false}, \ldots, n_{true}, n_{false}$, corresponding to possible truth assignments to the variables. For every $1 \leq i \leq n$, there are transitions, labeled $\#$, from x_i to i_{true} and i_{false}. On the fourth level there is a single accepting state q_{acc}. For every $1 \leq i \leq n$, value $val \in \{true, false\}$, and letter $1 \leq j \leq m$, there is a transition labeled j from i_{val} to q_{acc} iff assigning val to variable i satisfies the clause c_j. For example, if the literal $\neg x_5$ appears in clause c_2, then there is a transition labeled 2 from the state 5_{false} to q_{acc}. It is easy to see that the language of $\mathcal{A}^\theta$ is $\{j\#k : 1 \leq j \leq m, 1 \leq k \leq m\}$. The DFA $\mathcal{U}^\theta$ is such that $L(\mathcal{U}^\theta) = \{j\#j : 1 \leq j \leq m\}$. It is easy to define $\mathcal{U}^\theta$ with $m+2$ states. It is left to show that θ is satisfiable iff $\mathcal{A}^\theta$ is $\mathcal{U}^\theta$-DBP. Note that since $L(\mathcal{U}^\theta) \subseteq L(\mathcal{A}^\theta)$, the relaxed-DBP problem in our case amounts to deciding whether $\mathcal{A}^\theta$ has an embedded DFA $\mathcal{A}'$ such that $L(\mathcal{U}^\theta) \subseteq L(\mathcal{A}')$.

Assume first that θ is satisfiable. Let $f : \{1, ..., n\} \to \{true, false\}$ be a satisfying assignment to the variables of θ. We describe a deterministic automaton $\mathcal{A}'_f$ embedded in $\mathcal{A}^\theta$ such that $L(\mathcal{U}^\theta) \subseteq L(\mathcal{A}'_f)$. Note that in order to obtain from $\mathcal{A}^\theta$ a DFA, one should resolve two kinds of nondeterministic choices. First, for every state x_i, we have to choose a single $\#$-successor. We define the $\#$-successor of x_i to be $i_{f(x_i)}$. Then, in the initial state q_0, we have to choose for every letter $1 \leq j \leq m$ a single j-successor. Since f is a satisfying assignment for θ, then every clause c_j has at least one literal satisfied by f. For a letter j, we define the j-successor of q_0 to be x_i, for the minimal i such that either x_i is a literal in c_j and $f(x_i) =$ true of $\neg x_i$ is a literal in c_j and $f(x_i) = false$. It is easy to see that for all $1 \leq j \leq m$, the word $j\#j$ is in $L(\mathcal{A}'_f)$.

For the other direction, assume that $\mathcal{A}^\theta$ has an embedded DFA $\mathcal{A}'$ such that $L(\mathcal{U}^\theta) \subseteq L(\mathcal{A}')$. The transition function δ' of $\mathcal{A}'$ induces an assignment f where for all $1 \leq i \leq n$, we have that $f(i) =$ true iff

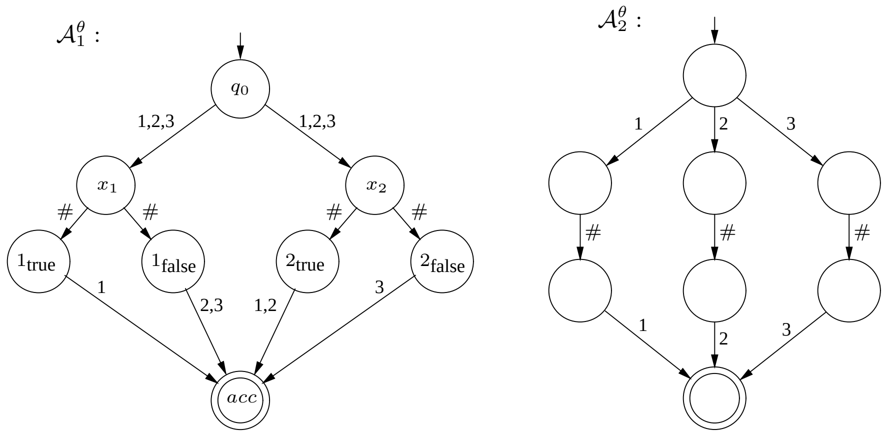

Fig. 1. The NFA $\mathcal{A}_1^\theta$ and the DFA $\mathcal{A}_2^\theta$ corresponding to $\theta = (x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_2) \wedge (\neg x_1 \vee \neg x_1 \vee \neg x_2)$.

$\delta'(x_i, \#) = i_{true}$. Since $L(\mathcal{U}^\theta) \subseteq L(\mathcal{A}')$, then for all $1 \leq j \leq m$, the word $j\#j$ is accepted by $\mathcal{A}'$. Thus, for every $1 \leq j \leq m$, there exists a variable $1 \leq i_j \leq n$, such that $\delta'(q_0, j) = x_{i_j}$, $\delta'(x_{i_j}, \#) = (i_j)_{val}$, and $\delta'((i_j)_{val}, j) = q_{acc}$. By the definition of $\mathcal{A}^\theta$, there is a transition labeled j from the state i_{val} to q_{acc} iff assigning val to x_i satisfies the clause c_j. It follows that for every clause c_j, the variable x_{i_j} is such that x_{i_j} is a literal in c_j and $\delta'(x_{i_j}, \#) = (i_j)_{true}$ or $\neg x_{i_j}$ is a literal in c_j and $\delta'(x_{i_j}, \#) = (i_j)_{false}$. Hence, f is a satisfying assignment for θ. $\square$

By Theorem 2, the application of our results to online algorithms is as follows.

Corollary 1. *Consider an optimization problem P with a set S of configurations, an approximation factor $\alpha \geq 1$, and an NFA $\mathcal{U}$.*

- **[model checking]** *Given an online algorithm g for P that uses configurations in S, deciding whether g is α-competitive with respect to environments restricted to input sequences in $L(\mathcal{U})$ can be solved in time polynomial in S and $\mathcal{U}$.*
- **[synthesis]** *Deciding whether there is an α-competitive online algorithm for P that uses configurations in S, with respect to environments restricted to input sequences in $L(\mathcal{U})$, is NP-complete.*

4 Reasoning about Online Algorithms with Look-ahead

In this section we describe a framework for reasoning about online algorithms that have a *bounded lookahead* on the requests yet to come. We consider the case where the online algorithm can see not only the next request, but rather the next $l + 1$ requests for some constant $l \geq 0$. For several classes of optimization problems, like dynamic location and online graph problems, it was shown that online algorithms with a lookahead above a certain minimal length can achieve better competitive ratios than algorithms with a shorter (or no) lookahead [CGS89,Ira94]. To the best of our knowledge, there are also problems, like online bipartite matching [KT91], for which it is not fully known how beneficial a lookahead can be.

An *online algorithm with lookahead l* for an optimization problem P is an algorithm that at each point $i > 0$ in time, reads the next $l + 1$ requests $r_i, \ldots, r_{i+l}$ that need to be processed, and serves the request r_i. The requests $r_{i+1}, \ldots, r_{i+l}$ (i.e., the *lookahead*) are *not* served at time i, but rather when their respective times come. The use of the lookahead at time i is only to guide the algorithm in serving the request r_i.[4] Formally, given a set Σ of requests, and a set Γ of actions, let $\perp$ be a new symbol designating

[4] Note that while this is perhaps the most natural kind of lookahead, other types of lookahead have also been considered in the literature. However, these (for example, the "strong lookahead" of [Alb97] for paging) are usually specifically tailored for a specific class of optimization problems.

the end of the input. A word $x = x_1 \cdots x_n \in (\Sigma \cup \{\perp\})^+$ is *legal* if for all $1 \leq j < n$, if $x_j = \perp$ then $x_{j+1} = \perp$. For $n > 0$, we denote by $\Sigma_\perp^n = \{x \in (\Sigma \cup \{\perp\})^n : x \text{ is legal}\}$ the set of all legal lookahead words of length n, and by $\Sigma_\perp^+$ the set $\bigcup_{n>0} \Sigma_\perp^n$ of all legal words in $(\Sigma \cup \{\perp\})^+$. An online algorithm with lookahead l corresponds to a function $g : \Sigma^+ \times \Sigma_\perp^l \to \Gamma$. The processing of a sequence of requests $\sigma_1 \cdots \sigma_n \in \Sigma^n$ by g is then $g(\sigma_1, \sigma_2 \cdots \sigma_{l+1}), g(\sigma_1\sigma_2, \sigma_3 \ldots \sigma_{l+2}), \ldots, g(\sigma_1 \ldots \sigma_n, \sigma_{n+1} \ldots \sigma_{n+l})$, where $\sigma_i = \perp$ for every $i > n$. Note that at time $i > 0$ the lookahead is $\sigma_{i+1} \cdots \sigma_{i+l}$, and it contains the end-of-input symbol for every position after the last request σ_n. Similar to the case with no lookahead, we say that an online algorithm with lookahead of length l uses a finite memory S, if there is a regular mapping of $\Sigma^* \times \Sigma_\perp^l$ to S such that g behaves in the same manner on identical continuations of words that are mapped to the same configuration. The definitions of the cost of processing a sequence of requests, as well as the definitions of α-competitiveness and the competitive ratio of g, are carried over from the definitions given in Section 2 for online algorithms with no lookahead.

In order to handle algorithms with lookahead, we construct (instead of the automaton $\mathcal{A}_P$ of Section 2.3) an automaton $\mathcal{A}_{P,l}$ such that every online algorithm for P that uses memory S and lookahead of length l is embedded in $\mathcal{A}_{P,l}$. The construction of $\mathcal{A}_{P,l}$ is very similar to that of $\mathcal{A}_P$, the main difference being that now the alphabet of $\mathcal{A}_{P,l}$ is $\Sigma \times \Sigma_\perp^l$, to match the way requests are presented to an online algorithm with a lookahead of length l. Observe that not all sequences of letters in $\Sigma \times \Sigma_\perp^l$ need be considered. Indeed, if $(\sigma, y), (\sigma', y') \in \Sigma \times \Sigma_\perp^l$ are two consecutive blocks of requests presented to the online algorithm, then it must be that $y = \sigma' \cdot y'_1 \cdots y'_{l-1}$, i.e., that the lookahead y indeed matches the following l requests. In order to make sure that irrelevant sequences have no influence, $\mathcal{A}_{P,l}$ does not accept such sequences (in fact, it simply crashes when reading such a sequence). To this end, $\mathcal{A}_{P,l}$ has to remember the lookahead in every input letter that it reads.

Formally, $\mathcal{A}_{P,l} = \langle \Sigma \times \Sigma_\perp^l, S_0 \cup (S \times \Sigma_\perp^l), \Delta, c, S_0, S \times \{\perp^l\}\rangle$, where $S_0 \subseteq S$ is the subset of initial configurations of S; For a source state u of the form $u = s \in S_0$ or $u = (s, x) \in S \times \Sigma_\perp^l$, an input $(\sigma, y) \in \Sigma \times \Sigma_\perp^l$, and a destination state $(s', x') \in S \times \Sigma_\perp^l$, we have that $\langle u, (\sigma, y), (s', x')\rangle \in \Delta$ iff *(i)* $y = x'$, and if u is of the form $u = (s, x)$ then $x_1 = \sigma$ and $x_2 \cdots x_l \cdot x'_l = y$, *(ii)* the set $\Gamma' \subseteq \Gamma$ of actions that process the request σ from configuration s, by updating the configuration to s', is non-empty; the cost of such a transition is $c(\langle u, (\sigma, y), (s', x')\rangle) = \min_{\gamma \in \Gamma'} \mathit{action\_cost}(\gamma)$; Note that the accepting states are all configurations that are coupled with a lookahead of $\perp^l$, which indicates that the input sequence has ended.

Let P be an optimization problem, and let $\mathcal{A}_{P,l}$ be a WFA for its algorithms that use memory S and lookahead of length l. Observe that, like $\mathcal{A}_P$, the automaton $\mathcal{A}_{P,l}$ represents the optimal offline algorithm for P in the sense that given a finite sequence of requests $w \in \Sigma^*$, the cost of w in $\mathcal{A}_P$ is the cost of w in an optimal offline algorithm whose configurations are in S. On the other hand, it is not hard to see that an online algorithm with lookahead of length l, that uses memory S, corresponds to a DWFA embedded in $\mathcal{A}_{P,l}$. Formally, given such an online algorithm $g : \Sigma^+ \times \Sigma_\perp^l \to \Gamma$, the DWFA embedded in $\mathcal{A}_{P,l}$ and induced by g is an automaton $\mathcal{A}_{P,l}^g$ in which, for every configuration $s \in S$, and every request (with lookahead) $(\sigma, y) \in \Sigma \times \Sigma_\perp^l$, we have that $\delta(s, (\sigma, y)) = (s', y)$ for every initial configuration $s \in S_0$, and $\delta(\langle(s, \sigma \cdot y_1 \cdots y_{l-1}), (\sigma, y)\rangle) = (s', y)$ for all $s \in S$; where s' is the configuration obtained by applying the action $g(w \cdot \sigma, y)$ from s, and w is such that $h(w, \sigma \cdot y_1 \cdots y_{l-1}) = s$.

Theorem 5. *Given an optimization problem P, a set S of configurations, and $l \geq 0$. Let $\mathcal{A}_{P,l}$ be a WFA that models online algorithms for P that use memory S and lookahead of length l. An online algorithm g that uses memory S and lookahead of length l is α-competitive iff $\mathcal{A}_{P,l}^g$ α-approximates $\mathcal{A}_{P,l}$.*

By [AKL10], given $\mathcal{A}_{P,l}$, deciding if it has an embedded DWFA that α-approximates it (and also obtaining such DWFAs) can be done in time polynomial in the size of $\mathcal{A}_{P,l}$ if $\alpha = 1$, and is NP-complete for $\alpha > 1$; whereas given $\mathcal{A}_{P,l}^g$, deciding if it α-approximates $\mathcal{A}_{P,l}$ can be done in polynomial time for all values of α. Thus, Theorem 5 implies the following:

Corollary 2. *Consider an optimization problem P with a set S of configurations, an approximation factor $\alpha \geq 1$, and some $l \geq 0$.*

- **[model checking]** *Given an online algorithm g for P that uses configurations in S and lookahead of size l, deciding whether g is α-competitive can be solved in time polynomial in S and exponential in l.*
- **[synthesis]** *Deciding whether there is an α-competitive online algorithm for P that uses memory S and lookahead of length l, can be done in polynomial deterministic (nondeterministic) time in S for $\alpha = 1$ ($\alpha > 1$, respectively) and time exponential in l.*

Note that the model-checking and synthesis algorithms that we get are exponential in l. While we do not prove a matching lower bound, we were able to prove co-NP-hardness in l (by a reduction from the problem of deciding whether an NFW accepts all words of length l or less). Also, earlier work on lookahead in ω-regular games suggests that an exponential cost in the lookahead cannot be avoided [HKT10].

5 Symbolic Model-Checking Algorithm

In this section we describe a symbolic model-checking algorithm for online algorithms. The explicit algorithm of [AKL10] gets as input a WFA $\mathcal{A}_1 = \langle \Sigma, Q_1, \Delta_1, c_1, S_1, F_1 \rangle$, a DWFA $\mathcal{A}_2 = \langle \Sigma, Q_2, \Delta_2, c_2, s_2, F_2 \rangle$ embedded in $\mathcal{A}_1$, and an approximation factor α, and decides in polynomial time whether $\mathcal{A}_2$ α-approximates $\mathcal{A}_1$.

Let $m = |Q_1| = |Q_2|$. The algorithm is based on iteratively calculating functions $f_i : Q_1 \times Q_2 \to \mathbb{Z} \cup \{-\infty, \infty\}$. The dependency in m is reflected both in the size of the required data structure, and the number of iterations that the algorithm performs. A symbolic algorithm cannot avoid the time complexity that the iterative calculation involves, but it copes with the space complexity by working with a symbolic representation of all the components of the automata and of the functions f_i.

The data structures we work with are *Binary Decision Diagrams* (BDDs, for short) [Bry86] and *multi-valued* BDDs (MVBDDs, for short). While a BDD represents a Boolean function, MVBDDs assign to each truth assignment of the variables a value in $\mathbb{Z} \cup \{-\infty, \infty\}$. We implement an MVBDD by an array of BDDs, each encoding a single bit of the value. Using b BDDs, the value of the MVBDD is then a b-bit signed two's complement integer. It has a minimum value of -2^{b-1} and a maximum value of $2^{b-1} - 1$ (inclusive). In addition, we maintain two BDDs, for $-\infty$ and ∞.

We now move to a detailed description of the symbolic model-checking algorithm (Figure 2). In addition to α, the algorithm gets as input a symbolic representation of $\mathcal{A}_1$ and $\mathcal{A}_2$. The sets of variables X and W are used in order to encode Q_1 and Σ, respectively. Accordingly, the transition function Δ_1 is described by an MVBDD $\mathit{trans}_1 : X \times W \times X' \to \mathbb{N} \cup \infty$, where X' is a tagged copy of X. Formally, $\mathit{trans}_1(\langle q_1, a, q_1' \rangle)$ is $c(\langle q_1, a, q_1' \rangle)$ for $\langle q_1, a, q_1' \rangle \in \Delta_1$, and is ∞ otherwise. Note that the domain of trans_1 are truth assignments to the variables in X, W, and X', and not tuples in $Q_1 \times \Sigma \times Q_1$; since, however, the variables encode such triples, we abuse notation and refer to $\mathit{trans}_1(\langle q_1, a, q_1' \rangle)$. Note that we use weights in $\mathbb{N}$ rather than in $\mathbb{R}^{\geq 0}$. The sets S_1 and F_1 are described by the BDDs init_1 and fin_1 over X, respectively. The WFA $\mathcal{A}_2$ is described in a similar manner, with variables in Y and W. Let $V = X \cup X' \cup W \cup Y \cup Y'$. For convenience, we refer to all BDDs as functions from V (even though the function they maintain may be independent of some of the variables).

The algorithm uses the following operators.

- The functions $\mathbf{not} : BDD \to BDD$ and $\mathbf{and} : BDD \times BDD \to BDD$ operate as the corresponding logical operators of negation and conjunction, respectively.
- The operator $\mathbf{set\_value}$ gets an MVBDD f, a BDD *cond*, and a value *val*, and sets the value of f to val for the inputs characterized by *cond*.

– The operator $\mathbf{prime}$ gets an MVBDD f whose function is independent of X' and Y' and turns it into an MVBDD that corresponds to the function obtained from f by replacing the variables in X and Y by their tagged copies in X' and Y'.
– The functions $\mathbf{add}, \mathbf{sub}, \mathbf{max}$: MVBDD $\times$ MVBDD $\to$ MVBDD return the MVBDD obtained by applying addition, subtraction, and maximum, respectively, on the given MVBDDs.
– The function $\mathbf{get\_BDD}$: MVBDD $\to$ BDD returns a BDD whose value is 1 exactly on the inputs on which the value of the given MVBDD is not ∞. In particular, $\mathbf{get\_BDD}$(*trans*) returns a BDD representing the (un-weighted) transitions.
– The function $\mathbf{less\_than}$: MVBDD $\times$ MVBDD $\to$ BDD gets two MVBDDs, f and g, and returns a BDD h such that for all $v \in 2^V$, we have $h(v) = 1$ iff $f(v) \leq g(v)$.
– The function $\mathbf{var\_max}$: $2^V \times$ MVBDD $\to$ MVBDD gets a set U of variables and an MVBDD g and returns an MVBDD f such that for all $v \in 2^V$, we have $f(v) = \max\{g(v') : v'$ agrees with v on the variables not in $U\}$. Note that $f(v)$ is independent of the variables in U. The function $\mathbf{cond\_max}$: $2^V \times$ BDD $\times$ MVBDD $\to$ MVBDD is similar, but gets in addition a BDD s, and the maximum of the MVBDD g is taken only over v''s that agree with v on the variables not in U and satisfy $s(v') = 1$. If no such v' exists, then $f(v) = -\infty$.

The algorithm calculates functions $f_i : X \times Y \to \mathbb{Z} \cup \{-\infty, \infty\}$, for $0 \leq i \leq 2m^2$. The function f_i indicates the competitiveness of $\mathcal{A}_2$ with respect to words of length at most i. Formally, for every two states $q_1 \in Q_1$ and $q_2 \in Q_2$, the value $f_i(q_1, q_2)$, for $0 \leq i \leq m^2$, equals $-\infty$ if no word of length at most i is accepted from q_1, it equals ∞ if there exists a word of length at most i that is accepted from q_1 but not from q_2, and it equals $t \in \mathbb{Z}$ if t is the maximal value such that there exists a word of size at most i that is accepted from q_1 at a cost of c, and from q_2 at a cost of $\alpha \cdot c + t$. For $m^2 \leq i \leq 2m^2$, the algorithm takes into account cycles along which the performance of $\mathcal{A}_1$ is "unboundedly better" than that of $\mathcal{A}_2$, in which case the value of $f_i(q_1, q_2)$ is increased to ∞. As proved in [AKL10], such cycles would be detected after at most m^2 iterations, and their influence on the ability of $\mathcal{A}_2$ to α-approximate $\mathcal{A}_1$ would be detected after another round of m^2 iterations. Thus, the algorithm needs not compute f_i for $i > 2m^2$.

The algorithm first defines f_0 so that $f_0(q_1, q_2)$ is $-\infty$ if $q_1 \notin F_1$, is 0 if $q_1 \in F_1$ and $q_2 \in F_2$, and is ∞ if $q_1 \in F_1$ and $q_2 \notin F_2$. Each loop iteration gets f_{i-1} and calculates f_i. For that, the algorithm calculates an MVBDD g. After executing Line 8, we have $g(\langle q_1, q_1', a, q_2, q_2' \rangle) = f_{i-1}(q_1', q_2') + c_2(q_2, a, q_2') - \alpha \cdot c_1(q_1, a, q_1')$. Thus, after Line 14, we have $f_i(q_1, q_2) = \max\{f_{i-1}(q_1, q_2), \max_{a \in \Sigma} f_a(q_1, q_2)\}$, where $f_a(q_1, q_2) = \max_{q_1' \in \delta_1(q_1, a)} [f_{i-1}(q_1', \delta_2(q_2, a)) + c_2(q_2, a, \delta_2(q_2, a)) - \alpha \cdot c_1(q_1, a, q_1')]$. If $i \geq m^2$ and $f_i(q_1, q_2) \geq f_{i-1}(q_1, q_2)$, then $f_i(q_1, q_2)$ is further increased, in Line 17, to ∞.

Finally, note that the fact we only care about embedded DWFA (only DWFA correspond to deterministic online algorithms) is crucial for the correctness of the algorithm. Indeed, the calculation of $f_a(q_1, q_2)$ makes use of the fact that in a DWFA, the state q_1 has only a single a-successor.

When implementing the symbolic algorithm, we have tried to minimize the maximal number of variables for a single MVBDD, but (as is the case with other symbolic algorithms that relate two systems) we could not avoid the construction of the MVBDD g that depends on all the variables in V.

We note here that the implementation of the symbolic algorithm is applicable also for the results appearing in the previous sections. The algorithm given in Section 3 for deciding whether a given online algorithm is α-competitive with respect to a given restriction on the environment actually uses the algorithm described above as a sub-routine. Before running the algorithm it should only compute a product of two automata. This can be easily implemented symbolically. As for the algorithm given in Section 4 for reasoning about online algorithms with lookahead, it simply uses the algorithm described above as a black-box.

Experimental Results Before describing our experimental results, we would like to stress that the main contribution of the paper is the ideas behind the algorithm – our implementation is not a suggestion

```
**Symbolic model-checking** $(init_1, trans_1, fin_1, init_2, trans_2, fin_2, \alpha)$

1: $\mathbf{set\_value}(f_0, \mathbf{not}(fin_1), -\infty)$;
2: $\mathbf{set\_value}(f_0, \mathbf{and}(fin_1, fin_2), 0)$;
3: $\mathbf{set\_value}(f_0, \mathbf{and}(fin_1, \mathbf{not}(fin_2)), \infty)$;
4: $i := 0$;
5: repeat
6:    $i++$;
7:    $\mathbf{prime}(f_{i-1})$;
8:    MVBDD $g := \mathbf{sub}(\mathbf{add}(f_{i-1}, trans_2), \alpha \cdot trans_1)$;
9:    BDD $t_1 := \mathbf{get\_BDD}(trans_1)$;
10:   BDD $t_2 := \mathbf{get\_BDD}(trans_2)$;
11:   BDD $match\_trans := \mathbf{and}(t_1, t_2)$;
12:   MVBDD $f_a := \mathbf{cond\_max}(D, match\_trans, g)$;
13:   MVBDD $h := \mathbf{var\_max}(W, f_a)$;
14:   $f_i := \mathbf{max}\{f_{i-1}, h\}$;
15:   if $i \geq m^2$ then
16:      BDD $diff := \mathbf{less\_than}(f_{i-1}, h)$;
17:      $\mathbf{set\_value}(f_i, diff, \infty)$;
18:   end if
19: until $(f_i == f_{i-1})$ or $(i == 2m^2)$;
20: BDD $init\_states := \mathbf{and}(init_1, init_2)$;
21: MVBDD $approx := \mathbf{var\_max}(X \cup Y, init\_states, f_i)$;
22: if $approx < \infty$ then
23:    return true;
24: else
25:    return false;
26: end if
```

Fig. 2. The symbolic model-checking algorithm.

for a ready-to-run tool, but rather a justification for the argument that our algorithm can actually be implemented symbolically. It is very likely that researchers with more experience in implementations could have come up with a much better implementation. We still find it encouraging that even our naive implementation has led us to interesting and practical insights, as described below.

The most natural modeling of paging is by a WFA whose set of states corresponds to the configurations of the cache. Such a modeling corresponds to non-marking algorithms, as it does not allow the algorithm to use information beyond the set of pages that are currently in the cache. In the course of applying our implementation of the symbolic algorithm to paging, we have realized that the only non-marking competitive algorithm for paging we are aware of, Flush-when-Full (FWF) [BEY98,KMRS88], is not lazy (also referred to as "demand paging" in [BEY98]); that is, it may evict from the cache more than a single page in case an eviction is required. From a practical standpoint, such evictions are wasteful, and a reasonable implementation of FWF would keep the cache full at all times and only mark the pages spuriously evicted by FWF – thus treating FWF as a marking algorithm. This has led us to the development of an online algorithm that is both lazy and non-marking. Unfortunately for us, we later discovered that this algorithm already appears as ROTATE in [CKPV91], where it is proved to be k-competitive, by means of amortized analysis. Below we give a brief description of FWF and ROTATE.

Algorithm FWF **(Flush-when-Full).** The idea behind FWF is quite simple: on a fault, if the cache is full, simply empty the cache, and then bring the requested page into the cache. Using our formalism, Algorithm FWF can be described as follows. Initially, the cache configuration is empty. At any given step, let C be the current configuration and r the requested page. Then

$$\mathsf{FWF}(C,r) = \begin{cases} C & \text{if } r \in C \\ C \cup \{r\} & \text{if } r \notin C \text{ and } |C| < k \\ \{r\} & \text{if } r \notin C \text{ and } |C| = k \end{cases}$$

Algorithm ROTATE**.** As mentioned in Section 5, it turns out that ROTATE is the only known k-competitive algorithm that is both lazy and non-marking.

Assume that the set of pages is $\Sigma = \{1, 2, ..., n\}$. We think of the sequence $1, 2, ..., n$ as being wrapped around a circle, say clockwise. If a requested page r is not in the cache, and the cache is full, find the page v in the cache that is closest, in the counter-clockwise order around the circle, to r, and replace v by r in the cache. Intuitively, the k cache locations travel clockwise around the circle in response to the requests, without passing each other.

We now give a more formal description. For $x, y \in \Sigma$, define

$$[x,y] = \begin{cases} \{x, x+1, ..., y\} & \text{if } x \le y, \\ \{x, x+1, ..., n, 1, ..., y\} & \text{if } x > y. \end{cases}$$

Initially, the cache configuration is empty. At any step, let C be the current cache configuration and r the new request. Then

$$\mathsf{ROTATE}(C,r) = \begin{cases} C & \text{if } r \in C, \\ C \cup \{r\} & \text{if } r \notin C \text{ and } |C| < k, \\ (C - \{v\}) \cup \{r\} & \text{if } r \notin C,\ |C| = k \text{ and } C \cap [v,r] = \{v\}. \end{cases}$$

We have studied the k-competitiveness of ROTATE and FWF using an implementation of the symbolic algorithm written in Java, using JavaBDD [Wha] as a high level object oriented layer, on top of the BDD library CUDD [Som]. As our test platform, we used a Pentium 4, 3.2Ghz Linux Machine with 4GB of RAM. Due to the limitations of the 32bit Java Virtual Machine we used, our program was limited to using at most 2GB of memory. Our experimental results are described in Figure 3. Note that the number of states in a WFA representing a paging algorithm with parameters n and k is of order n^k. Thus, the computational bottleneck is k. With our naïve implementation, this resulted in a memory requirement of over 2GB for very low values of k. It is also interesting to note that in some cases an explicit version of the algorithm actually performed better than the symbolic one. For example, the explicit version was able to prove the k-competitiveness of FWF with parameters $k = 4$ and $n = 2^3$ (8 iterations, 5.5 minutes, 650MB), as well as to model check ROTATE with parameters $k = 3$ and $n = 2^3$ (7 iterations, 3 minutes, 200MB). Both of these tests could not be completed in the symbolic implementation as it exceeded the 2GB memory limit. The explanation for the high memory requirement of the symbolic algorithm lies in the fact that some of the intermediate calculations require storing functions over five domains, whereas the explicit algorithm never has to consider functions over more than three domains. On the other hand, one can see that the symbolic algorithm could handle instances (like FWF with $k = 2$ and $n = 2^{54}$) that are clearly beyond the scope of an explicit implementation.

The experimental results achieved with our naive implementation are not impressive, but we find them encouraging. First, they prove that formal reasoning about competitive ratios of online algorithms is feasible, both in theory and practice. Second, even though the instances we considered were very small, they have led us to rediscover the algorithm ROTATE, showing that a lot of insight can be gained even when working with small instances. Third, we discovered that while in the worst case the symbolic algorithm may run for $2m^2$ iterations, in practice it converges many orders of magnitude faster. For

k	n	Time	Memory	# of Iterations
2	2^3	$<$ 1 Sec.	$<$ 100 MB	4
2	2^6	4 Sec.	170 MB	4
2	2^{20}	25 Sec.	600 MB	4
2	2^{40}	100 Sec.	1300 MB	4
2	2^{54}	6 Min.	2000 MB	4
3	2^2	$<$ 1 Sec.	$<$ 100 MB	6
3	2^3	10 Min.	620 MB	6

FWF

k	n	Time	Memory	# of Iterations
2	2^3	$<$ 1 Sec.	$<$ 100 MB	4
2	2^6	7 Sec.	170 MB	4
2	2^{10}	100 Sec.	250 MB	4
2	2^{20}	12 Hrs.	1300 MB	4

ROTATE

Fig. 3. Experimental Results

example, while some of our experiments have a value of $2m^2$ above 2^{50}, in all cases the algorithm converged to termination in at most 6 iterations! In fact, the main bottleneck seems to be the memory requirements of our BDD based implementation, and the associated time required to handle very big BDDs. It is our belief that representing MVBDDS not by arrays of BDDs, but rather by utilizing more efficient constructs such as *Multi Terminal BDDs (MTBDDs)* [FMY97], or *Algebraic Decision Diagrams (ADDs)* [BFG+97], would enable much larger instances to be handled.

Acknowledgment We thank Marek Chrobak for helpful discussions.

References

[AKL10] B. Aminof, O. Kupferman, and R. Lampert. Reasoning about online algorithms with weighted automata. *ACM Transactions on Algorithms*, 6(2), 2010.

[AKL11] B. Aminof, O. Kupferman, and R. Lampert. Rigorous Approximated Determinization of Weighted Automata. In *Proc. 26th IEEE Symp. on Logic in Computer Science*, pages 345–354, 2011.

[Alb97] S. Albers. On the influence of lookahead in competitive paging algorithms. *Algorithmica*, 18:283–305, 1997.

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDBK+94] S. Ben-David, A. Borodin, R.M. Karp, G. Tardos, and A. Wigderson. On the power of randomization in on-line algorithms. *Algorithmica*, 11(2):2–14, 1994.

[BEY98] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[BFG+97] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2-3):171–206, 1997.

[BIRS95] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive Paging with Locality of Reference. *Journal of Computer and System Sciences*, 50(2):244 – 258, 1995.

[Bre98] D. Breslauer. On competitive on-line paging with lookahead. *Theoretical Computer Science*, 209(1–2):365–375, 1998.

[Bry86] R.E. Bryant. Graph-based algorithms for Boolean-function manipulation. *IEEE Transactions on Computing*, C-35(8):677–691, 1986.

[CCH+05] A. Chakrabarti, K. Chatterjee, T.A. Henzinger, O. Kupferman, and R. Majumdar. Verifying quantitative properties using bound functions. In *Proc. 13th Conf. on Correct Hardware Design and Verification Methods*, volume 3725 of *Lecture Notes in Computer Science*, pages 50–64. Springer, 2005.

[CGS89] F. R. K. Chung, R. L. Graham, and M. E. Saks. A dynamic location problem for graphs. *Combinatorica*, 9(2):111–131, 1989.

[CHJ08] K. Chatterjee, T. Henzinger, and B. Jobstmann. Environment assumptions for synthesis. In *Proc. 19th Int. Conf. on Concurrency Theory*, volume 5201 of *Lecture Notes in Computer Science*, pages 147–161. Springer, 2008.

[CKPV91] M. Chrobak, H.J. Karloff, T.H. Payne, and S. Vishwanathan. New Results on Server Problems. *SIAM J. Discrete Math.*, 4(2):172–181, 1991.

[CL92] M. Chrobak and L.L. Larmore. The server problem and on-line games. In *On-line Algorithms*, volume 7 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 11–64, 1992.

[FMY97] M. Fujita, P.C. McGeer, and J.C.-Y. Yang. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design*, 10(2-3):149–169, 1997.

[Fra86] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer, 1986.

[HKT10] M. Holtmann, L. Kaiser, and W. Thomas. Degrees of lookahead in regular infinite games. In *Proc. 13th Int. Conf. on Foundations of Software Science and Computation Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 252–266. Springer, 2010.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Science Institutes*, pages 477–498. Springer, 1985.

[Imm88] N. Immerman. Nondeterministic space is closed under complement. *Information and Computation*, 17:935–938, 1988.

[Ira94] S. Irani. Coloring inductive graphs on-line. *Algorithmica*, 11(1):53–72, 1994.

[KMRS88] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[KS86] W. Kuich and A. Salomaa. *Semirings, Automata, Languages*, volume 5 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1986.

[KT91] M.-Y. Kao and S. R. Tate. Online matching with blocked input. *Inf. Process. Lett.*, 38(3):113–116, 1991.

[MMS90] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11(2):208–230, 1990.

[Moh97] M. Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[Pnu85] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Science Institutes*, pages 123–144. Springer, 1985.

[Som] F. Somenzi. CUDD package, release 2.4.1. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[ST85] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[Wha] J. Whaley. JavaBDD package, release 1.0b2. `http://javabdd.sourceforge.net/`.

[You91] N. Young. On-line caching as cache size varies. In *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms*, pages 241–250, 1991.