

DIVING INTO THE QUEUE

SIMON BEIER, MARTIN KUTRIB, ANDREAS MALCHER*
AND MATTHIAS WENDLANDT

Abstract. We introduce and study the model of diving queue automata which are basically finite automata equipped with a storage medium that is organized as a queue. Additionally, two queue heads are provided at both ends of the queue that can move in a read-only mode inside the queue. In particular, we consider suitable time constraints and the case where only a finite number of turns on the queue is allowed. As one main result we obtain a proper queue head hierarchy, that is, two heads are better than one head, and one head is better than no head. Moreover, it is shown that the model with one queue head, finitely many turns, and no time constraints as well as the model with two queue heads, possibly infinitely many turns, and time constraints is captured by P and has a P -complete membership problem. We obtain also that a subclass of the model with two queue heads is already captured by logarithmic space. Finally, we consider decidability questions and it turns out that almost nothing is decidable for the model with two queue heads, whereas we obtain that at least emptiness and finiteness are decidable for subclasses of the model with one queue head.

Mathematics Subject Classification. 68Q45, 68Q15.

Accepted November 21, 2018.

1. INTRODUCTION

The investigation of finite automata was probably one fundamental starting point in the theory of formal languages and automata. This model has widely been investigated from different vantage points both theoretical and practical. Finite automata feature many positive and desirable properties such as, for example, the equivalence of nondeterministic and deterministic models, the existence of minimization algorithms, the closure under many operations, and decidable questions such as emptiness, inclusion, or equivalence (see, *e.g.*, [13]). But on the other hand, their computational capacity is rather limited since only regular languages are accepted. To increase the computational capacity several storage media have been added such as pushdown stores [6], stack stores [8], queue tapes [3, 5, 24], or Turing tapes. The difference between pushdown stores and stack stores is that the latter model possesses an additional read-only head which may move inside the pushdown store and may compare pushdown contents with the input several times without deleting them. Pushdown automata are often used in the context of compilers checking the syntax of programming code, and the generalized model of stack automata is also applied to compiling questions in [9]. In contrast to finite automata without storage media such as deterministic or nondeterministic finite automata, some decidable questions such as inclusion,

Keywords and phrases: Queue automata, logspace Turing machines, formal languages, decidability questions.

Institut für Informatik, Universität Giessen, Arndtstr. 2, 35392 Giessen, Germany.

* Corresponding author: andreas.malcher@informatik.uni-giessen.de

equivalence, or universality get lost for (nondeterministic) pushdown or stack automata, but the questions of emptiness and finiteness remain decidable.

The model of a *queue* automaton where in comparison with pushdown automata the data structure of a pushdown store is replaced by a queue obeying a *first-in-first-out* principle and its investigation from a formal language point of view has not gained much attention in the literature yet. Some examples are [3, 5, 24] and some recent results on the variants of reversible queue automata and input-driven queue automata may be found in [16–18]. A recent survey may be found in [19]. In general, queue automata are very powerful since they can simulate Turing machines [24] and, hence, accept every recursively enumerable language. Thus, suitable restrictions to obtain more manageable models have to be considered. One such restriction constrains the available time to (quasi-)realtime [5]. However, questions such as emptiness, finiteness, or equivalence are still undecidable for realtime queue automata. Another restriction bounds the number of alternating enqueueing and dequeuing phases to a finite number and yields so-called finite-turn queue automata. It is shown in [16] that emptiness and finiteness are decidable for such automata.

In this paper, we translate the extension from pushdown automata to stack automata to queue automata with the additional constraints of quasi-realtime and finite-turn boundedness. In analogy to stack automata, we provide two read-only heads, one at the front of the queue and one at the tail of the queue, which may move inside the queue. We call such automata *diving queue automata* for which it is then possible to compare queue contents with the input several times without deleting them and, since there are two heads, to compare some part of the queue with some other part of the queue. We will also consider the restricted model where only one head is available. The paper is organized as follows. In Section 2 we define the different variants of diving queue automata and present some example languages that can be accepted by finite-turn diving queue automata with one or two queue heads. A hierarchy on the number of heads is obtained in Section 3 using the unary languages $\{a^{n^2} \mid n \geq 0\}$ and $\{a^{n^3} \mid n \geq 0\}$. It turns out that the first language shows that one head is better than no head, whereas the latter language gives that two heads are more powerful than one head. In Section 4, we investigate the computational capacity of finite-turn queue automata in more detail. We obtain that the family of languages accepted by finite-turn queue automata with one head, as well as the family of languages accepted by weakly quasi-realtime queue automata with two heads, is included in the complexity class P and has a P-complete membership problem. We also show that a subclass of the family of languages accepted by weakly quasi-realtime finite-turn queue automata with two heads is already included in the complexity class L. On the other hand, every realtime deterministic pushdown or stack automaton can be simulated by a queue automaton having two heads and performing no turn, whereas every realtime deterministic counter automaton can be simulated by a queue automaton having only one head and performing no turn. Finally, we study in Section 5 decidability questions for finite-turn queue automata. The main results are that in case of two queue heads and no turns even the question of emptiness is undecidable. On the other hand, if only one head is available it is possible to identify some subclasses for which at least the questions of emptiness and finiteness are decidable.

2. PRELIMINARIES

We write Σ^* for the set of all *words* over the finite alphabet Σ . The *empty word* is denoted by λ , and $\Sigma^+ = \Sigma^* \setminus \{\lambda\}$. For the *reversal* of a word w we write w^R , and for the *length* of w we write $|w|$. For convenience, we use Σ_λ for $\Sigma \cup \{\lambda\}$. We use \subseteq for *inclusions* and \subset for *strict inclusions*.

A *queue automaton* is a system consisting of a finite state control, a one-way input tape, and a data structure *queue*. At each time step, it is possible to remove or keep the symbol at the front and to possibly enter symbols at the end of the queue. The transition depends on the current state, the symbols being currently at the front and end of the queue, the case whether or not an input symbol is read, and, in the former case, on the current input symbol. Often queue automata are defined so that they can only see the symbol at the front of the queue. However, with an eye towards generalizations we extend the definition as mentioned. It should be noted that the additional knowledge of the last queue symbol does not increase the computational power of queue automata.

A device can simply store the symbol lastly entered at the end of the queue in its state to simulate the behavior of the extended automaton.

Here, we study *diving queue automata* that are classical queue automata with the additional ability to move the queue heads from the front and/or end inside the queue without altering the contents. Additionally, the automaton can neither append symbols to the end of the queue nor remove symbols from the front of the queue while at least one head is inside the queue. In this way, it is possible to read but not to change the stored information. To enable this behavior, the symbol at the front of the queue is marked by index \triangleright , the symbol at the end of the queue by index \triangleleft , and a single symbol in the queue by index \boxtimes .

A *deterministic diving queue automaton*, abbreviated as DDQA, is a system $\langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$, where

1. Q is the finite set of *internal states*,
2. Σ is the finite set of *input symbols*,
3. Γ is the finite set of *queue symbols*,
4. for $x \in \{\triangleright, \triangleleft, \boxtimes\}$, the set Γ_x is a marked copy of Γ , that is, $\Gamma_x = \{a_x \mid a \in \Gamma\}$, and $\bar{\Gamma}$ is the union $\Gamma \cup \Gamma_{\triangleright} \cup \Gamma_{\triangleleft} \cup \Gamma_{\boxtimes}$,
5. $q_0 \in Q$ is the *initial state*,
6. $\perp \notin \Gamma$ is the *empty-queue symbol*,
7. $F \subseteq Q$ is the set of *accepting states*, and
8. δ is a (partial) *transition function* from $Q \times \Sigma_\lambda \times ((\bar{\Gamma} \times \bar{\Gamma}) \cup (\{\perp\} \times \{\perp\}))$ to $Q \times \{\mathbf{remove}, -1, 0, +1\} \times (\Gamma^+ \cup \{-1, 0, +1\})$. Whenever $\delta(p, x, z_1, z_2) = (q, \alpha_1, \alpha_2)$ is defined, then
 - (a) $\alpha_1 \in \{\mathbf{remove}\}$ only if $(z_1, z_2) \in (\Gamma_{\triangleright} \times \Gamma_{\triangleleft}) \cup (\Gamma_{\boxtimes} \times \Gamma_{\boxtimes})$,
 - (b) $\alpha_2 \in \Gamma^+$ only if $(z_1, z_2) \in (\Gamma_{\triangleright} \times \Gamma_{\triangleleft}) \cup (\Gamma_{\boxtimes} \times \Gamma_{\boxtimes}) \cup (\{\perp\} \times \{\perp\})$.

In particular, there must never be a choice of using an input symbol or of using λ input. So, it is required that for all p in Q and z_1, z_2 in $\bar{\Gamma} \cup \{\perp\}$: if $\delta(p, \lambda, z_1, z_2)$ is defined, then $\delta(p, a, z_1, z_2)$ is undefined for all a in Σ .

The interpretation of a transition $\delta(p, x, z_1, z_2) = (q, \alpha_1, \alpha_2)$, where $p \in Q$ is the current state, $x \in \Sigma_\lambda$ is the current input symbol read or λ , $z_1 = z_2 = \perp$, if the queue is empty, and $z_1, z_2 \in \bar{\Gamma}$ are the symbols being currently at the front and end of the queue if it is not empty, is as follows.

1. State q is entered.
2. $\alpha_1 \in \{\mathbf{remove}\}$ means to remove the symbol at the front of the queue.
3. $\alpha_2 \in \Gamma^+$ is the word to be entered at the end of the queue.
4. $\alpha_1 = -1$ means to move the front head one square to the left, $\alpha_1 = 0$ means to keep it at the current square, and $\alpha_1 = +1$ means to move it one square to the right.
5. $\alpha_2 = -1$ means to move the tail head one square to the left, $\alpha_2 = 0$ means to keep it at the current square, and $\alpha_2 = +1$ means to move it one square to the right.

A *configuration* of a DDQA $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$ is a tuple (q, w, u, p_1, p_2) , where $q \in Q$ is the current state, $w \in \Sigma^*$ denotes the unread part of the input, $u \in \Gamma_{\triangleright} \Gamma^* \Gamma_{\triangleleft} \cup \Gamma_{\boxtimes} \cup \{\lambda\}$ is the current queue content, where the leftmost symbol is at the front, and $0 \leq p_1, p_2 \leq |u| + 1$ are the current positions of the queue heads (see Fig. 1), where it is understood that a head at position 0 or $|u| + 1$ is out of the queue and scans nothing.

In order to implement the marking of the queue contents a mapping φ is used, that maps a string of two or more symbols from $\bar{\Gamma}^+$ to a string from $\Gamma_{\triangleright} \Gamma^* \Gamma_{\triangleleft}$ by keeping the symbols and adjusting the indices appropriately. A string of length one is mapped to the corresponding symbol from Γ_{\boxtimes} .

The *initial configuration* for input w is set to $(q_0, w, \lambda, 0, 0)$. During the course of its computation, M runs through a sequence of configurations. One step from a configuration to its successor configuration is denoted by \vdash .

Let $q, q' \in Q$, $a \in \Sigma_\lambda$, $w \in \Sigma^*$, $z \in \Gamma_{\boxtimes}$, $z_1 z_2 \cdots z_m \in \Gamma_{\triangleright} \Gamma^* \Gamma_{\triangleleft}$, for $m \geq 2$, $1 \leq p_1, p_2 \leq m$, $v \in \Gamma^+$, and $d_1, d_2 \in \{-1, 0, 1\}$. We set

1. $(q, aw, \lambda, 0, 0) \vdash (q', w, \lambda, 0, 0)$, if $\delta(q, a, \perp, \perp) = (q', 0, 0)$,
2. $(q, aw, \lambda, 0, 0) \vdash (q', w, \varphi(v), 1, |v|)$, if $\delta(q, a, \perp, \perp) = (q', 1, v)$,

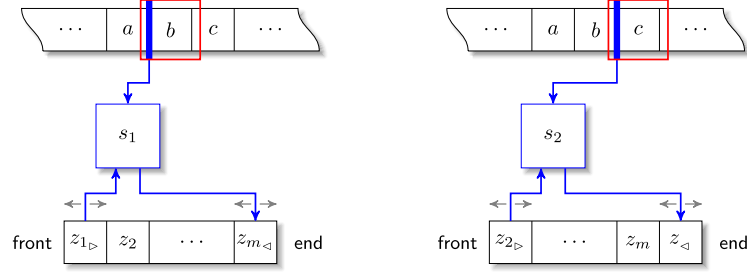


FIGURE 1. Successive configurations of a DDQA, where $\delta(s_1, b, z_{1\triangleright}, z_{m\triangleleft}) = (s_2, \mathbf{remove}, z)$.

3. $(q, aw, z, 1, 1) \vdash (q', w, \varphi(v), 1, |v|)$, if $\delta(q, a, z, z) = (q', \mathbf{remove}, v)$,
4. $(q, aw, z, 1, 1) \vdash (q', w, \lambda, 0, 0)$, if $\delta(q, a, z, z) = (q', \mathbf{remove}, 0)$,
5. $(q, aw, z, 1, 1) \vdash (q', w, \varphi(zv), 1 + d_1, |zv|)$, if $\delta(q, a, z, z) = (q', d_1, v)$,
6. $(q, aw, z, 1, 1) \vdash (q', w, z, 1, 1)$, if $\delta(q, a, z, z) = (q', 0, 0)$,
7. $(q, aw, z_1 z_2 \cdots z_m, 1, m) \vdash (q', w, \varphi(z_2 z_3 \cdots z_m v), 1, m - 1 + |v|)$, if $\delta(q, a, z_1, z_m) = (q', \mathbf{remove}, v)$,
8. $(q, aw, z_1 z_2 \cdots z_m, 1, m) \vdash (q', w, \varphi(z_2 z_3 \cdots z_m), 1, m - 1 + d_2)$, if $\delta(q, a, z_1, z_m) = (q', \mathbf{remove}, d_2)$,
9. $(q, aw, z_1 z_2 \cdots z_m, 1, m) \vdash (q', w, \varphi(z_1 z_2 \cdots z_m v), 1 + d_1, m + |v|)$, if $\delta(q, a, z_1, z_m) = (q', d_1, v)$,
10. $(q, aw, z_1 z_2 \cdots z_m, p_1, p_2) \vdash (q', w, z_1 z_2 \cdots z_m, p_1 + d_1, p_2 + d_2)$, if $\delta(q, a, z_{p_1}, z_{p_2}) = (q', d_1, d_2)$.

For all other situations, \vdash is undefined and M halts. Whenever the queue is empty, the successor configuration is computed by the transition function with the special empty-queue symbol \perp . We denote the reflexive and transitive closure of \vdash by \vdash^* and the transitive closure of \vdash by \vdash^+ . The language accepted by the DDQA M is the set $L(M)$ of words for which the computation beginning in the initial configuration halts in a configuration in which the whole input is read and an accepting state is entered:

$$L(M) = \{w \in \Sigma^* \mid (q_0, w, \lambda, 0, 0) \vdash^* (q, \lambda, u, p_1, p_2) \text{ with some } q \in F, \\ 0 \leq p_1, p_2 \leq |u| + 1, \text{ and } M \text{ halts in } (q, \lambda, u, p_1, p_2)\}.$$

In general, the family of all languages that are accepted by some device X is denoted by $\mathcal{L}(X)$.

In the sequel, we distinguish two further modes of moving the queue heads into the queue. If only the head from the front of the queue may move inside, the devices are called *deterministic front diving queue automata* (DfDQA), and if only the head from the tail of the queue may move inside, we call the devices *deterministic tail diving queue automata* (DtDQA). It is straightforward that both types of devices have the same computational capacity. For example, a DfDQA can be simulated by a DtDQA as follows. Whenever the DfDQA moves its front queue head into the queue, the DtDQA moves its tail queue head through the queue to the front end, and subsequently simulates the DfDQA using its tail head to simulate the front head of the DfDQA. Similarly, a DtDQA can be simulated by a DfDQA. So, we may use the abbreviation D1DQA for DfDQA as well as for DtDQA. For simplicity we omit the straightforward adaptations of the definitions for these variants. *Classical deterministic queue automata* without the possibility to move the queue heads into the queue are abbreviated with DQA.

It is well known that general deterministic queue automata are computationally universal, that is, they can simulate Turing machines [24]. So, in the sequel we also consider restricted variants. More precisely, we limit the maximal number of λ -steps on each square of the input tape that can be performed *when the queue heads are not inside the queue*. A DDQA is said to be *weakly quasi realtime* if there is a constant that bounds this number for all computations. The DDQA is said to be *weakly realtime* if this constant is 0, that is, if there are no such λ -steps at all.

For a computation of a (diving) queue automaton, a *turn* is a phase in which the length of the queue first increases and then decreases. There may occur two cases. Formally, a *turn* is either a step of the form

$$(q_1, w_1, z_1 z_2 \cdots z_m, 1, m) \vdash (q_2, w_2, z_2 z_3 \cdots z_m v, 1, m - 1 + |v|),$$

where δ removes the first symbol from the queue and enters the word v with $|v| \geq 1$ into the queue at the same time, or a sequence of at least three configurations

$$(q_1, w_1, u_1, p_{1,1}, p_{2,1}) \vdash (q_2, w_2, u_2, p_{1,2}, p_{2,2}) \vdash \cdots \vdash (q_n, w_n, u_n, p_{1,n}, p_{2,n}),$$

where $|u_1| < |u_2| = \cdots = |u_{n-1}| > |u_n|$ and no step in between removes from and enters into the queue. For any given $k \geq 0$, a *k-turn* computation is any computation containing exactly k turns.

A DDQA performing *at most* k turns in *any* computation is called *k-turn DDQA* and will be denoted by DDQA_k . The same notation is also used for the other variants of (diving) queue automata. A *k-turn* queue automaton processes its input in at most $2k + 1$ phases alternating between enqueueing and dequeuing of symbols. We will call such phases *enqueueing* phases and *dequeuing* phases, respectively.

In order to illustrate our definitions we continue with some examples.

Example 2.1. The Gladkij language $L_1 = \{w\$w^R\$w \mid w \in \{a, b\}^*\}$ is accepted by a deterministic tail diving queue automaton M_1 with no turns. M_1 reads the input up to the first $\$$ and enqueues every symbol read. Thus w is stored in the queue. Then M_1 reads the second part of the input w^R while it checks with its tail head from the tail of the queue to the front symbolwise whether the current symbol of the input is the same as the symbol stored in the queue. If there is a mismatch, then M_1 rejects. Otherwise, M_1 reads the second $\$$ and moves its tail head from the front of the queue back to the tail while checking in the same way that the last part of the input equals w . The Gladkij language is known to be not growing context-sensitive [4] and, hence, is also not a Church-Rosser language [21].

Example 2.2. The language $L_2 = \{(ba^n)^n \mid n \geq 1\}$ is accepted by a deterministic diving queue automaton M_2 using no turn. For $n = 1$, M_2 can check using its states whether the input is ba . Given an input word w with $n > 1$, then M_2 enqueues the longest prefix of the form ba^+ , that is, ba^n . Subsequently, M_2 uses its front head to check that the number of b 's in the remaining input is exactly $n - 1$. To this end, it positions the front head on the leftmost a and moves one position to the right whenever a b is read in the input. The tail head is used to check that every a -block in the input is exactly of length n . To this end, it positions the tail head on the rightmost a and moves one position to the left whenever an a is read in the input. After n a 's the tail head scans b_{\triangleright} , M_2 has to read a b while moving the front head, and in the check of the next a -block the tail head moves one position to the right whenever an a is read in the input. After n a 's the tail head scans a_{\triangleleft} , M_2 has to read a b , and changes again its direction. Finally, M_2 enters an accepting state when the front head scans a_{\triangleleft} and the tail head scans a_{\triangleleft} or b_{\triangleright} after reading at least one a . It should be noted that L_2 is not an indexed language [1, 7] and thus is not accepted by any nested stack automaton [2].

Example 2.3. The non-semilinear language $L_3 = \{a^{n^2} \mid n \geq 0\}$ can be accepted by a deterministic tail diving queue automaton M_3 with no turns. The basic idea of the construction is based on the fact that n^2 can be represented as the sum $n^2 = \sum_{i=1}^n (2i - 1)$. The idea is now that M_3 starts with one A in the queue and then iteratively performs the following phases: M_3 uses its tail head to read for every A in the queue an a in the input, the tail head returns to the tail of the queue, and two additional A 's are enqueued. In this way, M_3 reads one a in the first phase, a^3 in the second phase, and in general a^{2i-1} in the i th phase. Altogether, $1 + 3 + 5 + \cdots + (2i - 1) = i^2$ many a 's have been read after the i th phase.

Formally, $M_3 = \langle \{q_0, q_1, q_2, q_3\}, \{a\}, \{A\}, \delta, q_0, \perp, \{q_1\} \rangle$. In the first step, M_3 enqueues three A 's using the rule $\delta(q_0, \lambda, \perp, \perp) = (q_1, 1, AAA)$. Then the queue contents is $A_{\triangleright} AA_{\triangleleft}$. Thus, there is one non-marked symbol in the queue when M_3 is in state q_1 for the first time.

State q_1 is the only accepting state of M_3 and the first word accepted is $a^0 = \lambda$. In every following phase, M_3 moves its tail head from the tail to the front of the queue and for every non-marked A in the queue, one symbol a of the input is read. When the head reaches the front, it moves back to the tail, two more A 's are enqueued, and the accepting state q_1 is entered. If there are a 's left in the input, the next phase is started. The detailed rules are as follows.

$$\begin{aligned} \delta(q_1, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q_1, 0, -1), & \delta(q_2, \lambda, A_{\triangleright}, A_{\triangleright}) &= (q_3, 0, 1), \\ \delta(q_1, a, A_{\triangleright}, A) &= (q_2, 0, -1), & \delta(q_3, \lambda, A_{\triangleright}, A) &= (q_3, 0, 1), \\ \delta(q_2, a, A_{\triangleright}, A) &= (q_2, 0, -1), & \delta(q_3, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q_1, 0, AA). \end{aligned}$$

Since M_3 is in the accepting state q_1 after the i th phase is completed and input a^{i^2} has been read, M_3 accepts language L_3 .

3. TWO-PARTY DIVING IS BETTER THAN LONESOME DIVING

In this section, we will separate the language families accepted by finite-turn queue automata having two, one, or no queue heads. First, we present a technical result on the role played by λ -steps when the queue heads are not inside the queue which will be used in the sequel.

Proposition 3.1. *For every weakly quasi-realtime diving queue automaton an equivalent weakly realtime diving queue automaton can effectively be constructed.*

Proof. Let $k \geq 1$ be a constant and $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$, be a weakly quasi-realtime DDQA that does not perform more than k λ -steps on every input square while the queue heads are not inside the queue.

In order to sketch the construction of an equivalent weakly realtime DDQA M' , we first consider that the queue heads do not move inside the queue while M performs a sequence of λ -steps. Then the construction is along the lines of several similar proofs (see, e.g., [5] or [15]). Roughly, during at most k consecutive λ -steps the DDQA M can access at most k symbols at the front of the queue. To simulate these k steps at once, the queue symbols of M' are k -fold compressed queue symbols of M . Moreover, M' maintains a buffer in its finite control that allows to store up to k queue symbols of M . This buffer stores the symbols at the front of the queue of M . Whenever the buffer gets empty, M' removes the symbol at the front of the queue and stores it in the buffer. In this way, M' can simulate k operations at the front of the queue at once. Similarly, M' maintains another buffer in its finite control that is used to store at most k queue symbols from the end of the queue of M . Whenever this buffer gets full, its content is entered as compressed symbol into the queue and the buffer is emptied. In this way, M' can store exclusively compressed symbols at the tail of the queue. Clearly, it can simulate k operations at the tail of the queue at once. So far, M' can simulate up to k consecutive λ -steps at once unless these steps move the queue heads into the queue.

If M moves the queue heads into the queue upon reading an input symbol, M' simulates the step directly.

Next, we sketch the behavior of M' while its queue heads are inside the queue. The problem to cope with is that M may perform some λ -steps that count for the constant k , then by another λ -step it may move the heads into the queue where it performs λ -steps that do not count, and subsequently the heads may move to the queue ends where the λ -steps count again, and so on. In order to overcome such situations, M' behaves as follows if its heads are inside the queue and have to be moved. The idea is that both heads are not moved to the end of the queue simultaneously. So, if both heads are inside the queue one is moved according to the simulated step. If it reaches the end of the queue, it is then moved back and the second head is moved according to the simulated step. In this way M' collects the information which queue symbols are read by M in the next step and whether both heads would be moved to the ends of the queue *without moving the heads to the ends of the queue*. Similarly, M' collects the information if only one of its heads is inside the queue. Finally, if the heads are not moved to the ends of the queue, M' simulates the next step of M and moves its heads accordingly. On the other hand, if the heads would both be moved to the ends of the queue, M' simulates this step only if subsequently no λ -step is performed by M . Otherwise, the following λ -steps with queue heads not inside the

queue are simulated at once until M moves its heads into the queue again or would read an input symbol. In the latter case we are done. In the former case the behavior is repeated *without moving the heads to the ends of the queue*. So, a sequence of λ -steps that may be performed by M in between or before or after moving its heads into the queue are simulated by M' by a sequence of λ -steps with its heads inside the queue, which do not count for weakly realtime. It is worth mentioning that to this end the buffers used by M' have to be extended depending on the precise transition function of M . Moreover, computations with short queue contents have to be treated separately with the help of the buffers. However, the buffer sizes are finite with respect to the definition of M . The tedious technical details are omitted. \square

Our next result concerns finite-turn D1DQA that accept unary languages. It is known owing to Example 2.3 that some D1DQA_0 can accept the non-semilinear unary language $\{a^{n^2} \mid n \geq 0\}$. On the other hand, it is shown in [16] that finite-turn DQA can only accept semilinear languages. Thus, we obtain the following corollary.

Corollary 3.2. *The family of languages accepted by finite-turn DQA is properly included in the family of languages accepted by finite-turn D1DQA.*

As next result we show that D1DQA accepting unary languages can be simulated by *deterministic one-way non-erasing stack automata*. Basically, a one-way stack automaton is a conventional pushdown automaton with one-way input tape and the additional ability to move the pushdown head starting from the top inside the pushdown store and to read the contents of the pushdown store. As it is the case for diving automata, the contents of the pushdown store cannot be changed when the head is inside the store. Details on this model are given in [8]. There is also the generalization of stack automata having a two-way input tape. Since this generalization is not relevant for our needs in this paper, we will tacitly assume that all variants of stack automata considered here have a one-way input tape. One variant of deterministic stack automata (DSA) are deterministic *non-erasing* stack automata (DNESA) which can never pop symbols from the pushdown store. We will later consider also their nondeterministic variants abbreviated by NSA and NNESSA. For the next result we provide the following technical lemma.

Lemma 3.3. *Let $k \geq 0$, $M = \langle Q, \{a\}, \Gamma, \delta, q_0, \perp, F \rangle$ be a D1DQA_k accepting a unary language, and m be the length of the longest word that M can enter into the queue in one step. Then after at most $2 \cdot (k+1)^3 \cdot m^2 \cdot |Q|^7 \cdot |\Gamma|^6$ computation steps, M does not make any further turn.*

Proof. A k -turn queue automaton processes its input in at most $2k+1$ phases alternating between enqueueing and dequeuing of symbols. First, we consider an enqueueing phase of M .

Since M can enter symbols into the queue only if the queue heads are not inside the queue, we determine the number of possibilities for such situations. A situation is described by the both symbols at the front and at the end of the queue, by the current state, and by the behavior of M when its queue head is moved into the queue. This behavior depends on the queue contents but can be described by the state that M enters when the queue head leaves the queue. So, we obtain no more than $|Q|^2 \cdot |\Gamma|^2$ different situations. If such a situation occurs twice in an enqueueing phase automaton M is in an infinite loop. We conclude that the queue length can grow to at most $(k+1) \cdot m \cdot |Q|^2 \cdot |\Gamma|^2$ symbols after $k+1$ enqueueing phases, unless M enters an infinite loop.

In order to determine the number of steps that M may compute in its enqueueing phases without entering a loop, we have to consider that M moves its queue head into the queue as well. Let u be the queue contents. Then the queue head can be inside the queue for at most $|u| \cdot |Q|$ steps without being in a loop. Since $|u|$ is bounded from above by $(k+1) \cdot m \cdot |Q|^2 \cdot |\Gamma|^2$, we obtain that the queue head can be inside the queue for at most $l = (k+1) \cdot m \cdot |Q|^3 \cdot |\Gamma|^2$ steps. Therefore, in total, the enqueueing phases of M take no more than $l \cdot (k+1) \cdot |Q|^2 \cdot |\Gamma|^2 = (k+1)^2 \cdot m \cdot |Q|^5 \cdot |\Gamma|^4$ steps.

In a dequeuing phase, again, there are at most $|Q|^2 \cdot |\Gamma|^2$ different situations that M can be in with its queue head not inside the queue, unless it enters an infinite loop. Considering the time steps in which the queue head dives, there are at most $|u| \cdot |Q| \cdot k \cdot |Q|^2 \cdot |\Gamma|^2 \leq (k+1)^2 \cdot m \cdot |Q|^5 \cdot |\Gamma|^4$ steps until M removes a symbol from the queue, unless M enters an infinite loop. So, the total number of steps taken by M in its dequeuing phases without being in a loop is $|u| \cdot (k+1)^2 \cdot m \cdot |Q|^5 \cdot |\Gamma|^4 \leq (k+1)^3 \cdot m^2 \cdot |Q|^7 \cdot |\Gamma|^6$.

Summing up, altogether there are at most $2 \cdot (k+1)^3 \cdot m^2 \cdot |Q|^7 \cdot |\Gamma|^6$ computation steps in enqueueing and dequeuing phases until M has performed its last turn. \square

Proposition 3.4. *Let $k \geq 0$ be a constant and M be a $D1DQA_k$ accepting a unary language. Then an equivalent DNEA can effectively be constructed.*

Proof. Let $M = \langle Q, \{a\}, \Gamma, \delta, q_0, \perp, F \rangle$ be a $D1DQA_k$. We will describe the construction of an equivalent DNEA M' .

Since M is deterministic and the input is unary, there exists an integer m (depending on M) such that, on input prefix a^m , automaton M enters a configuration c from which no further turn is performed. This integer m can be computed by the Lemma 3.3. Now, in order to obtain configuration c it suffices to simulate M until it has consumed m input symbols (or got into an infinite loop before, which can easily be detected), whereby all prefixes accepted are recorded. Let q be the state and u be the queue contents of c . The possible operations that M can perform on its queue after reaching configuration c can be divided into two cases. Which case applies can be determined from the pre-computation of M on input a^m .

In the first case, M stays for the remaining computation in a dequeuing phase and hence never enqueues a new symbol. Since the length of the queue is bounded by $|u|$, it is possible to keep the contents of the queue in the state set of a finite automaton. Then, we can simulate M by a finite automaton and therefore also by a DNEA M' .

In the second case, M stays for the remaining computation in an enqueueing phase and hence never dequeues a symbol. A DNEA M' simulating M can be described as follows. First, M' pushes the queue contents u onto its stack while reading a^m . This is possible in a deterministic way, since u and m are fixed. Moreover, M' can accept any prefix whose length is shorter than m . Additionally, M' has states that are pairs where it keeps in its two components of the state, the state q of M and the first symbol of u which is the bottom-most symbol in M' 's stack. To simulate the remaining computation on input a^n , we differentiate between situations in which M enqueues some new symbol and situations in which M 's queue head is inside the queue. To simulate the latter situations, we use the stack head of M' to realize queue head moves of M and update the current simulated state accordingly. For the former situations, M' pushes all symbols to be enqueued by M onto its stack and updates the current simulated state accordingly. M' simulates the behavior of M by using its stack. Since M cannot dequeue, the queue can be simulated by the stack and thus M' accepts $L(M)$. \square

In the next theorem, we will separate the family of languages accepted by finite-turn $D1DQA$ and finite-turn $DDQA$.

Theorem 3.5. *The family of languages accepted by finite-turn $D1DQA$ is a proper subset of the family of languages accepted by finite-turn $DDQA$.*

Proof. The claimed inclusion follows from structural reasons. By results in [23] it follows that the language $L = \{a^{n^3} \mid n \geq 0\}$ is not accepted by any deterministic stack automaton. By Proposition 3.4 we know that L is not accepted by any finite-turn $D1DQA$ either. Thus, the properness of the claimed inclusion is shown by constructing a $DDQA_0$ $M = \langle Q, \{a\}, \{A\}, \delta, q_0, \perp, \{q_0, q_f\} \rangle$ for language L . We set $Q = \{q_0, q'_{i,0}, q_f\} \cup \bigcup_{1 \leq i \leq 3} \{q_{l,i}, q_{p,i}, q_{r,i}, q'_{l,i}, q'_{p,i}\}$.

We describe the construction inductively. In the beginning of the computation, automaton M is in its initial state q_0 which is defined as an accepting state, since the empty word belongs to L . In the first step, M reads the first a , enqueues the string AAA , and enters the accepting state q_f , since the input a belongs to L as well. Thus, we have the rule $\delta(q_0, a, \perp, \perp) = (q_f, 1, AAA)$ and the queue contents $A_{\triangleright}AA_{\triangleleft}$, where the front head scans A_{\triangleright} and the tail head scans A_{\triangleleft} .

Now, let $x \geq 1$ and assume that M is in state q_f , has read input a^{x^3} , and has queue contents $A_{\triangleright}A^xA_{\triangleleft}$, where the front head scans A_{\triangleright} and the tail head scans A_{\triangleleft} . To enter q_f the next time on overall input $a^{(x+1)^3}$, we have to read $3x^2 + 3x + 1$ further a 's, since $(x+1)^3 = x^3 + 3x^2 + 3x + 1$.

First, to read one additional a we add the rule $\delta(q_f, a, A_{\triangleright}, A_{\triangleleft}) = (q'_{l,0}, 0, 0)$. Second, to read $3x$ additional a 's, the front head moves three times from the front of the queue to the tail, while for every occurrence of A in

the queue one input symbol is read. Thus, we add the rules

$$\begin{aligned}
 \delta(q'_{l,i}, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q_{l,i+1}, 1, 0), \text{ for } 0 \leq i \leq 2, \\
 \delta(q_{l,i}, a, A, A_{\triangleleft}) &= (q_{l,i}, 1, 0), \text{ for } 1 \leq i \leq 3, \\
 \delta(q_{l,i}, \lambda, A_{\triangleleft}, A_{\triangleleft}) &= (q'_{l,i}, -1, 0), \text{ for } 1 \leq i \leq 3, \\
 \delta(q'_{l,i}, \lambda, A, A_{\triangleleft}) &= (q'_{l,i}, -1, 0), \text{ for } 1 \leq i \leq 3, \text{ and} \\
 \delta(q'_{l,3}, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q'_{p,1}, 0, 0).
 \end{aligned}$$

Third, to read $3x^2$ additional a 's the DDQA M uses both heads. The tail head successively moves to the front of the queue, while for every A seen the front head reads all x symbols A in the queue and the input head reads a^x . In this way, exactly x^2 symbols a can be read. This procedure has to be repeated three times. Thus, we will consider $1 \leq i \leq 3$ in the following rules.

$$\begin{aligned}
 \delta(q'_{p,i}, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q_{p,i}, 1, -1), & \delta(q'_{p,i}, \lambda, A, A) &= (q'_{p,i}, -1, 0), \\
 \delta(q_{p,i}, a, A, A) &= (q_{p,i}, 1, 0), & \delta(q'_{p,i}, \lambda, A_{\triangleright}, A) &= (q_{p,i}, 0, -1), \\
 \delta(q_{p,i}, \lambda, A_{\triangleleft}, A) &= (q'_{p,i}, -1, 0), & \delta(q_{p,i}, \lambda, A_{\triangleright}, A) &= (q_{p,i}, 1, 0).
 \end{aligned}$$

If M is in the situation that both heads are at the front of the queue, the computation of x^2 is done and the tail head is moved back to the tail.

$$\begin{aligned}
 \delta(q_{p,i}, \lambda, A_{\triangleright}, A_{\triangleright}) &= (q_{r,i}, 0, 1), \\
 \delta(q_{r,i}, \lambda, A_{\triangleright}, A) &= (q_{r,i}, 0, 1).
 \end{aligned}$$

Finally, to connect the three procedures we add the following rules.

$$\begin{aligned}
 \delta(q_{r,i}, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q'_{p,i+1}, 0, 0), \text{ for } 1 \leq i \leq 2, \\
 \delta(q_{r,3}, \lambda, A_{\triangleright}, A_{\triangleleft}) &= (q_f, 0, A).
 \end{aligned}$$

When state q_f is entered, we add a new A to the queue for the computation of the next accepted input word. We observe that when M is in state q_f , then M has read input $a^{(x+1)^3}$ and has queue contents $A_{\triangleright}A^{x+1}A_{\triangleleft}$, where the front head scans A_{\triangleright} and the tail head scans A_{\triangleleft} . Thus, M accepts language L . Since no symbols are ever dequeued, the DDQA M performs no turn. \square

4. COMPUTATIONAL CAPACITY OF DIVING QUEUE AUTOMATA

The section is devoted to exploring the computational capacity of diving queue automata. We have already seen that finite-turn DDQA can accept languages that are neither accepted by stack automata nor by nested stack automata. Here, we first turn to show that, for all $k \geq 0$, the complexity class L , which is the class of languages accepted by deterministic Turing machines with logarithmic space bounds, is an upper bound of the capacity of quasi-realtime DQA_k . This result is subsequently extended to show that a subclass of the family of languages accepted by weakly quasi-realtime $DDQA_k$ is included in the complexity class L as well. Another result will be that every realtime deterministic pushdown or stack automaton can be simulated by some weakly realtime $DDQA_0$, whereas every realtime deterministic counter automaton can be simulated by some weakly realtime $D1DQA_0$. Finally, we obtain that both the family of languages accepted by $D1DQA_k$ without time constraints and the family of languages accepted by weakly quasi-realtime $DDQA$, belongs to the complexity class P and has a P -complete membership problem.

4.1. Queue automata with two diving heads

Proposition 4.1. *Let $k \geq 0$ be a constant. The family of languages accepted by quasi-realtime DQA_k is included in the complexity class L .*

Proof. Given a quasi-realtime DQA_k M we construct a deterministic Turing machine T with two-way read-only input tape and log-space bounded two-way read-write work tape that accepts $L(M)$. This shows that $L(M)$ belongs to L .

According to the proof of Proposition 3.1 we may assume that M works in realtime. Since M works in realtime, any enqueueing and dequeueing phase starts at a different position in the input. During the simulation of M by T , the Turing machine only maintains input positions and the current length of the queue on the work tape, which can be done in log space. Whenever necessary, T recomputes the symbol at the front of the queue.

In particular, T uses a counter to store the current length of the queue. In addition, it uses counters E_1, E_2, \dots, E_{k+1} to store the input positions at which the enqueueing phases start, counter I to store the current input position of the simulation, and a counter R that stores the position at which the recomputation of the symbol at the front of the queue continues. Initially, the current length of the queue is set to zero, counter E_1 and I are set to one, while the counters E_2, E_3, \dots, E_{k+1} are set to *undefined*. The resimulation will start at position two, *that is*, counter R is set to two. Apart from other information, the state of T can store the symbols which are currently at the front and at the tail of the queue, the current state of M , the last transition performed in the recomputation, *that is*, the transition which is performed by M when reading the input symbol at the position stored in counter R , and the transitions performed whenever a new enqueueing phase starts.

Now T starts to simulate the first enqueueing phase (see Fig. 2 for an example). For any step, the next input symbol is read as M does and the counter I for the current position in the input is increased by one. So, T can directly simulate a transition of M . Whenever a word is entered into the queue, T can update the information about the tail symbol and the current queue length accordingly.

When the following dequeueing phase starts, T proceeds as follows. If the queue operation is a combined operation that enters into and removes from the queue, first the entering as above and then the removing is simulated. For the simulation of a dequeueing operation, T checks whether the queue is non-empty by inspecting the counter maintaining the queue length. If the queue is empty, M tries to remove from the empty queue and would halt. In this case T halts as well. Otherwise, the counter storing the queue length is decreased by one and T starts to recompute the information about the next symbol at the front of the queue. To this end, T remembers the currently simulated state and related information in an additional state register and uses counter R and an auxiliary counter to move its input head back to the position stored in R . From there, T can continue the resimulation by using the corresponding transition information stored in its state.

If the next step of the resimulation enters a word into the queue then the word entered is the new front of the queue. So, counter R is increased by one, the transition performed for resimulating the step is stored into the state set of T , and by exploiting the counter I and using an auxiliary counter Turing machine T moves its input head back to the current input position, where the next step is simulated (the corresponding transition has been remembered in the state set).

Now assume that the next step of the resimulation removes a symbol from the queue. All queue operations that M performs up to the position stored in counter R have completely been resimulated. Since the queue contents has been checked to be non-empty, from the position in counter R up to the current position more symbols have been entered than removed by M . Moreover, all remove operations between the two positions have already been simulated by T . Therefore, the resimulation can be continued with the next enter operation, *that is*, at the position at which the next enqueueing phase starts. To this end, T uses counter E_2 to move its head to that position, copies the position into counter R , and uses the transition performed at the position of counter E_2 to resimulate the next enqueueing step (the transition has been remembered in the state set) as above.

Subsequent enqueueing and dequeueing phases are simulated in the same way by using the appropriate counters E_i . So, all at most $2k + 1$ phases of M are simulated and T accepts if and only M does.

At this place it is evident that the construction given so far is slightly more involved. This is due to the fact that M may enter a word, not only a single symbol, into the queue in one step. If this is the case, the

Transition	Queue contents
$\delta(q_0, a_1, \perp, \perp) = (q_1, 1, z_1)$	z_1
$\delta(q_1, a_2, z_1, z_1) = (q_2, 0, z_2)$	$z_1 z_2$
$\delta(q_2, a_3, z_1, z_2) = (q_3, 0, z_3)$	$z_1 z_2 z_3$
$\delta(q_3, a_4, z_1, z_3) = (q_4, \mathbf{remove}, 0)$	$z_2 z_3$
$\delta(q_4, a_5, z_2, z_3) = (q_5, \mathbf{remove}, 0)$	z_3
$\delta(q_5, a_6, z_3, z_3) = (q_6, 0, z_4)$	$z_3 z_4$
$\delta(q_6, a_7, z_3, z_4) = (q_7, 0, z_5)$	$z_3 z_4 z_5$
$\delta(q_7, a_8, z_3, z_5) = (q_8, \mathbf{remove}, 0)$	$z_4 z_5$

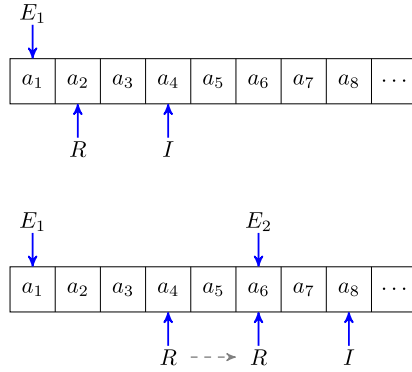


FIGURE 2. Eight transitions on input prefix $a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8$ performed by a DQA_k (top). Two situations of a simulating log-space bounded Turing machine (middle, bottom). In the first situation, the current transition to be simulated is on input symbol a_4 . In order to recompute the symbol z_2 at the front of the queue, the transition of the DQA_k on input symbol a_2 is resimulated. For the second situation, the current transition to be simulated is on input symbol a_8 . In order to recompute the symbol z_4 at the front of the queue, the resimulation continues at input symbol a_4 detecting that a dequeuing phase starts. Thus, the counter R is moved to the position at which the next enqueueing phase starts (E_2) and the resimulation continues from there.

resimulation reveals this word as well. However, T can store the word instead of the symbol in its state, and can use it symbol by symbol to simulate remove operations starting a resimulation only if the word has been consumed entirely. \square

We are able to extend the statement of Proposition 4.1 to a subclass of the family of languages accepted by weakly quasi-realtime $DDQA_k$. Within an enqueueing or dequeuing phase the queue heads may arbitrarily often enter the queue and move inside the queue, hence performing arbitrarily many queue head reversals, before they return to the ends of the queue and another queue operation takes place (a head reversal means that the direction of the head movement changes). Let $\ell \geq 0$ be a constant. Then, we call a $DDQA_k$ (∞, ℓ) -bounded ((∞, ℓ) - $DDQA_k$), if the total number of queue head reversals in every enqueueing phase may be unbounded, but the total number of queue head reversals in every dequeuing phase is bounded by ℓ . Similarly, a $DDQA_k$ is called (ℓ, ∞) -bounded ((ℓ, ∞) - $DDQA_k$), if the total number of queue head reversals in every dequeuing phase may be unbounded, but the total number of queue head reversals in every enqueueing phase is bounded by ℓ . Analogous definitions can be made for $D1DQA_k$ and are needed in Section 5.

Proposition 4.2. *Let $k, \ell \geq 0$ be constants. The family of languages accepted by weakly quasi-realtime (ℓ, ∞) -DDQA $_k$ is included in the complexity class L .*

Proof. Given a weakly quasi-realtime (ℓ, ∞) -DDQA $_k$ M we construct a deterministic Turing machine T with two-way read-only input tape and log-space bounded two-way read-write work tape that accepts $L(M)$. To do this, we extend the proof of Proposition 4.1. By Proposition 3.1 we may assume that M works in weakly realtime.

In addition to the proof of Proposition 4.1 the Turing machine T also has counters for the current positions of the queue heads and for the positions in the input, where M is in an enqueueing phase and has moved both queue heads to the ends of the queue again. Since M performs at most ℓ queue head reversals in every enqueueing phase, the number of such input positions is bounded by a constant. Moreover, the state of T stores the transition which is performed by M at such input positions. We have to add to the proof of Proposition 4.1 the behavior of T when M moves its queue heads into the queue during simulation or during resimulation.

First, we look at the case when the simulation is in a dequeuing phase and a symbol is removed from the front of the queue. Then, by resimulation the new symbol at the front of the queue has to be determined. In the proof of Proposition 4.1, it is shown how this is done when the queue heads do not move into the queue during resimulation. Here, we look at the case when the resimulation is in an enqueueing phase, the queue heads are at the ends of the queue, and at least one queue head moves into the queue in the current step. During this phase, where at least one queue head is inside the queue, the queue content is not changed, in particular there are no symbols entered into the queue. So, this phase does not have to be resimulated and we can continue with resimulation at the input position where the queue heads are back at the ends of the queue. This position is stored in some counter. So, the resimulation can continue appropriately with the corresponding transition stored in the state of T .

Finally, we have to consider the case when the queue heads are moved inside the queue during simulation of an enqueueing or dequeuing phase. By inspecting the counters for the current queue length and for the current positions of the queue heads in the queue, T can always detect whether the heads are at both ends of the queue and, thus, whether the transitions to be simulated work on marked queue symbols or not. Moreover, T can detect whether M moves a queue head out of the queue. Now, whenever M moves a queue head into the queue or within the queue, T has to determine the queue symbols at the current head positions, has to simulate the step of M , and to update the head position counters appropriately. So, it remains to be described how the queue symbols at the current head positions are determined. This is done by T similarly as the recomputation of the symbol at the front of the queue. Starting at the input position pointed to by c , the entire queue contents can successively be recomputed step by step by resimulation of the DDQA on input symbols following the position given in c where the dequeuing phases and the phases where the queue heads are inside the queue during enqueueing phases are ignored. Since the head position is given by a counter, T knows how many queue symbols have to be recomputed to obtain the one the head points to. During this recomputation the counter c is not changed. To this end, auxiliary counters are used as before. \square

The construction of the log-space bounded Turing machine in the proof of Proposition 4.2 reveals even more. We derive that, for all $k, \ell \geq 0$, the upper bound L cannot be reached.

Theorem 4.3. *Let $k, \ell \geq 0$ be constants. The family of languages accepted by weakly quasi-realtime (ℓ, ∞) -DDQA $_k$ is properly included in the complexity class L .*

Proof. It has been shown in [11] that the complexity class L is characterized by deterministic two-way multi-head finite automata. One part of the characterization is the simulation of a given log-space bounded Turing machine by a multi-head finite automaton. The construction shows that the number of heads necessary for the simulation depends on the cardinality of the tape alphabet of the Turing machine only.

Since the number of counters used and, thus, the cardinality of the tape alphabet in the proof of Proposition 4.2 depends on k and ℓ only, we may conclude that any weakly quasi-realtime (ℓ, ∞) -DDQA $_k$ can effectively be simulated by a deterministic two-way k' -head finite automaton, where the number k' of necessary heads depends only on k and ℓ .

On the other hand, in [22] it was proven that $k' + 1$ heads are better than k' . More precisely, for each $k' \geq 1$, there is a unary language accepted by some deterministic two-way finite automaton with $k' + 1$ heads which is not accepted by any deterministic two-way k' -head finite automaton. So, for all $k, \ell \geq 0$, there is a unary language belonging to \mathbf{L} that is not accepted by any weakly quasi-realtime (ℓ, ∞) -DDQA $_k$. \square

If we drop the constraint of a finite number of queue head reversals in every enqueueing phase, we obtain that the complexity class \mathbf{P} is an upper bound.

Theorem 4.4. *The family of languages accepted by weakly quasi-realtime DDQA is included in the complexity class \mathbf{P} .*

Proof. Due to Proposition 3.1 we only have to consider a weakly realtime DDQA M . Let $|Q|$ be the number of states of M and c be the length of the longest word that can be entered to the queue by M . We now consider a word w of length n that is accepted by M . Because M works in weakly realtime the length of the queue is at most cn during the whole computation of M on w . Having read an input symbol, the queue content is not changed until the next input symbol is read, since M is a weakly realtime device. So between the reading of two input symbols, M can perform at most $|Q| \cdot (cn)^2$ many steps, because for each of the two queue heads there are at most cn possible positions in the queue and a configuration cannot appear twice, because then M would perform a loop and would not accept w . There are n input symbols, so M performs at most $n \cdot |Q| \cdot (cn)^2 = |Q| \cdot c^2 n^3$ steps on the input w .

Now, M is simulated by a Turing machine T . If an input is accepted by M , the Turing machine T performs the simulation in polynomial time. Additionally, T has a counter that counts the length of the input and another counter that counts the number of steps that M has performed since the last input symbol was read. If this number is greater than $|Q| \cdot (cn)^2$, where n is the length of the input, M performs a loop and T rejects the input. Thus, T works in polynomial time and decides whether or not an input is accepted by M . \square

The next result shows that the class of realtime deterministic context-free languages is a lower bound for the capacity of weakly realtime DDQA $_0$.

Proposition 4.5. *For every realtime deterministic pushdown automaton an equivalent weakly realtime DDQA $_0$ can effectively be constructed.*

Proof. Let $A = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$ be a realtime deterministic pushdown automaton. The symbols have an analogous meaning as in the definition of DDQA. For deterministic pushdown automata, the transition function δ is a partial mapping from $Q \times \Sigma \times (\Gamma \cup \{\perp\})$ to $Q \times (\Gamma^+ \cup \{\$, 0\})$, where a word from Γ^+ means to push this word on the pushdown store, $\$$ means to pop the symbol from the top of the pushdown store, and 0 means that the pushdown store remains unchanged. We write δ_1 and δ_2 for the induced mappings from $Q \times \Sigma \times (\Gamma \cup \{\perp\})$ to Q and to $\Gamma^+ \cup \{\$, 0\}$, respectively. Rules of the form $\delta_2(q, a, \perp) = \$$ are not allowed.

We will now construct an equivalent weakly quasi-realtime DDQA $_0$ B . This proves the claim of the proposition, because by Proposition 3.1 an equivalent weakly realtime DDQA $_0$ can effectively be constructed. Formally, we define the DDQA $_0$ $B = \langle Q', \Sigma, \Gamma', \delta', (q_0, \perp), \perp, F \times (\Gamma \cup \{\perp\}) \rangle$, where $Q' = Q \cup Q \times (\Gamma \cup \{\perp\})$ and $\Gamma' = \Gamma \cup \{\$, \}$, and δ' is the partial mapping from $Q' \times \Sigma \times ((\overline{\Gamma'} \times \overline{\Gamma'}) \cup (\{\perp\} \times \{\perp\}))$ to $Q' \times \{\mathbf{remove}, -1, 0, +1\} \times ((\Gamma')^+ \cup \{-1, 0, +1\})$ defined as follows. Let $q \in Q$, $g \in \Gamma \cup \{\perp\}$, $a \in \Sigma$, $(e, f) \in (\overline{\Gamma'} \times \overline{\Gamma'}) \cup (\{\perp\} \times \{\perp\})$, and $\pi : \overline{\Gamma'} \cup \{\perp\} \rightarrow \Gamma' \cup \{\perp\}$ be the mapping that forgets the index of a queue symbol.

The first rule we have is

$$\delta'((q, g), a, e, f) = (\delta_1(q, a, g), d, \delta_2(q, a, g)), \text{ if } (e, f) \in (\Gamma'_{\triangleright} \times \Gamma'_{\triangleleft}) \cup (\Gamma'_{\boxtimes} \times \Gamma'_{\boxtimes}) \cup (\{\perp\} \times \{\perp\}),$$

where $d = 0$ if $\delta_2(q, a, g) = 0$ and $d = 1$ otherwise. Here, the state (q, g) of B says that A is in state q and has g as the symbol at the top of the pushdown store. This is why B starts in (q_0, \perp) . Now the input symbol a is read. Then $\delta_1(q, a, g)$ gives the new state of A . If $\delta_2(q, a, g) = g_1 g_2 \cdots g_m \in \Gamma^+$ then A pushes $g_1 g_2 \cdots g_m$ on the pushdown store and B enters $g_1 g_2 \cdots g_m$ into the queue. If $\delta_2(q, a, g) = \$$, then A performs the pop-operation

and B memorizes this by entering $\$$ at the end of the queue. If $\delta_2(q, a, g) = 0$, the pushdown store of A is not changed, so we also do not change the queue of B .

The next rule is used to determine the symbol on the top of the pushdown store of A . The idea is the following. The second head moves from the end of the queue to the left reading every symbol on its way. Whenever the second head reads a $\$$, the first head, which starts at the front of the queue, moves one step to the right, and whenever the second head reads a symbol different from $\$$, then the first head moves one step to the left. So the first head counts the difference between the number of $\$$ and the other symbols which are read by the second head on its way from the right to the left. The first head starts at the front of the queue with a difference of 0. So whenever the first head comes back to the front of the queue the difference is 0 again. If the second head is on a symbol different from $\$$ now, this symbol is on the top of the pushdown store of A , and B memorizes this by going to the state $(q, \pi(f))$ in the third row of the rule. This works whenever the pushdown store of A is not empty. If the pushdown store is empty, then the second head will move to the left until it reaches the front of the queue and reads a symbol different from $\$$ there. At this time the first head must be directly on the right of the second head, because the mentioned difference is 1 at that moment. In this situation B memorizes that the pushdown store of A is empty by the fourth row of the rule. So after applying the given rule multiple times, B knows now what is on the top of the pushdown store of A . The first head is at the front of the queue at this time again.

$$\delta'(q, \lambda, e, f) = \begin{cases} (q, +1, -1) & \text{if } \pi(f) = \$, \\ (q, -1, -1) & \text{if } \pi(f) \neq \$ \wedge e, f \in \Gamma' \cup \Gamma'_{\triangleleft}, \\ ((q, \pi(f)), 0, 0) & \text{if } \pi(f) \neq \$ \wedge e \in \Gamma'_{\triangleright} \cup \Gamma'_{\boxtimes} \cup \{\perp\}, \\ ((q, \perp), -1, 0) & \text{if } \pi(f) \neq \$ \wedge f \in \Gamma'_{\triangleright} \wedge e \in \Gamma' \cup \Gamma'_{\triangleleft}. \end{cases}$$

The last rule moves the second head back to the end of the queue:

$$\delta'((q, g), \lambda, e, f) = ((q, g), 0, +1) \quad \text{if } f \in \Gamma' \cup \Gamma'_{\triangleright}.$$

When the second head is at the end of the queue again, the next input symbol is read by the first rule. With these three rules B simulates the behavior of A . After reading the last symbol of the input with the first rule, B determines the symbol g on the top of the pushdown store of A one last time with the second rule, and moves the second head back to the end of the queue with the third rule. Then B halts in the state (q, g) , where q is the state which is reached by A after reading the whole input. The accepting states of B are $F \times (\Gamma \cup \{\perp\})$, so B accepts the input if and only if A accepts the input. This proves the proposition. \square

This result can be generalized from pushdown automata to stack automata.

Proposition 4.6. *For every realtime deterministic stack automaton an equivalent weakly realtime DDQA₀ can effectively be constructed.*

Proof. To prove this we extend the proof of Proposition 4.5. When the head of a realtime deterministic stack automaton A is at the top of the stack, then A can be simulated by a weakly quasi-realtime DDQA₀ B in the same way a realtime deterministic pushdown automaton is simulated. When the head of A is inside the stack, then the tail head of B should be at the same symbol as the head of A and the front head of B should be at the front of the queue. We need to explain what happens when the head of A moves one position up or down in the stack. In the finite control of B we remember if A ever processed a pop-operation. If this is not the case, the head moves of A can directly be simulated by the tail head of B . If A processed at least one pop-operation, then we will describe in the following how the head moves of A can be simulated by B . This is also illustrated in Figure 3.

When the head of A moves one position down, then the tail head of B moves one position to the left. Then it moves further to the left while the front head counts the difference between the number of $\$$ and the other

symbols which are read by the tail head on its way to the left, as in the proof of Proposition 4.5. When the front head is at the front of the queue again and the tail head is on a symbol different from \$, then this is the new symbol the head of A stands on.

Next, we have to consider the case where the head of A moves one position up. In this case, the tail head of B moves from its current position to the right until it reaches the tail of the queue. Meanwhile the front head starts at the front of the queue and counts the difference between the number of symbols different from \$ and the number of \$ symbols read by the tail head on its way to the right. When the tail head arrives at the tail, let the front head be at position i . This means that the difference between the number of symbols different from \$ and the number of \$ symbols read by the tail head on its way to the right equals $i - 1$. Thus the head of A is at position $i - 1$ in the stack, where the top is position 1. The new position of the head of A is $i - 2$. So we have $i - 2 \geq 1$, which gives $i \geq 3$. There are always at least i symbols in the queue of B , because A processed at least one pop-operation during its computation.

Now the front head of B moves two positions to the left and is at position $i - 2$. After that the tail head moves from the tail to the left while the front head moves one position to the right for every \$ symbol read by the tail head and the front head moves one position to the left for every other symbol read by the tail head. When the front head is at the front of the queue again and the tail head is at a symbol different from \$, then the difference between the number of symbols different from \$ and the number of \$ symbols read by the tail head on its way to the left is $(i - 2) - 1 = i - 3$. So the tail head of B is on the symbol in the queue that corresponds to the symbol at position $1 + (i - 3) = i - 2$ in the stack, which means that we have found the new symbol the head of A stands on. So we have described how the movements of the head of the stack automaton A can be simulated by the DDQA₀ B . This proves the proposition. \square

4.2. Queue automata with one diving head

Let us now study the class of D1DQA_k in more detail. First, we can show in analogy to Proposition 4.5 that the family of languages accepted by realtime deterministic counter automata is a lower bound for the capacity of weakly realtime D1DQA₀.

Proposition 4.7. *For every realtime deterministic counter automaton an equivalent weakly realtime D1DQA₀ can effectively be constructed.*

Proof. A deterministic counter automaton M is a deterministic pushdown automaton whose pushdown alphabet comprises one symbol, say Z , only. The basic idea for the simulation of M by some D1DQA M' is to indicate the current height of the counter by the position of M' 's tail head. Initially, some symbol Z' is enqueued while all other symbols enqueued are Z . Thus, possible queue contents are of the form $Z'_{\triangleright} Z^n Z_{\triangleleft}$ for $n \geq 0$ or Z'_{\triangleright} and have the following meaning:

- (1) If the tail head scans symbol Z'_{\triangleright} or $Z'_{\triangleright\triangleleft}$, the counter is zero.
- (2) If the tail head scans the ℓ th symbol Z from the left, the counter is ℓ .
- (3) If the tail head scans symbol Z_{\triangleleft} , the counter is $n + 1$.

To simulate a push-operation of some word Z^m , for $m \geq 1$, automaton M' updates its state and enqueues the word Z^m in case (3) or in case (1) when the tail head scans $Z'_{\triangleright\triangleleft}$. In case (2) or in case (1) when the tail head scans Z'_{\triangleright} , automaton M' moves its tail head m positions to the right, if $\ell + m \leq n + 1$. Otherwise, M' moves its tail head $n - \ell + 1$ positions to the right on symbol Z_{\triangleleft} and subsequently enqueues the word $Z^{m-(n-\ell+1)}$. To simulate a pop-operation in M , automaton M' updates its state and moves its tail head one position to the left in case (2) or in case (3), while M' blocks in case (1). Finally, M' only has to update its state when M only changes its state leaving its pushdown store unchanged.

So, M' can simulate M in this way and, moreover, M' never dequeues a symbol and hence performs no turn. Finally, M' works in weakly quasi realtime if M works in realtime. By Proposition 3.1 we may assume that M' works in weakly realtime. \square

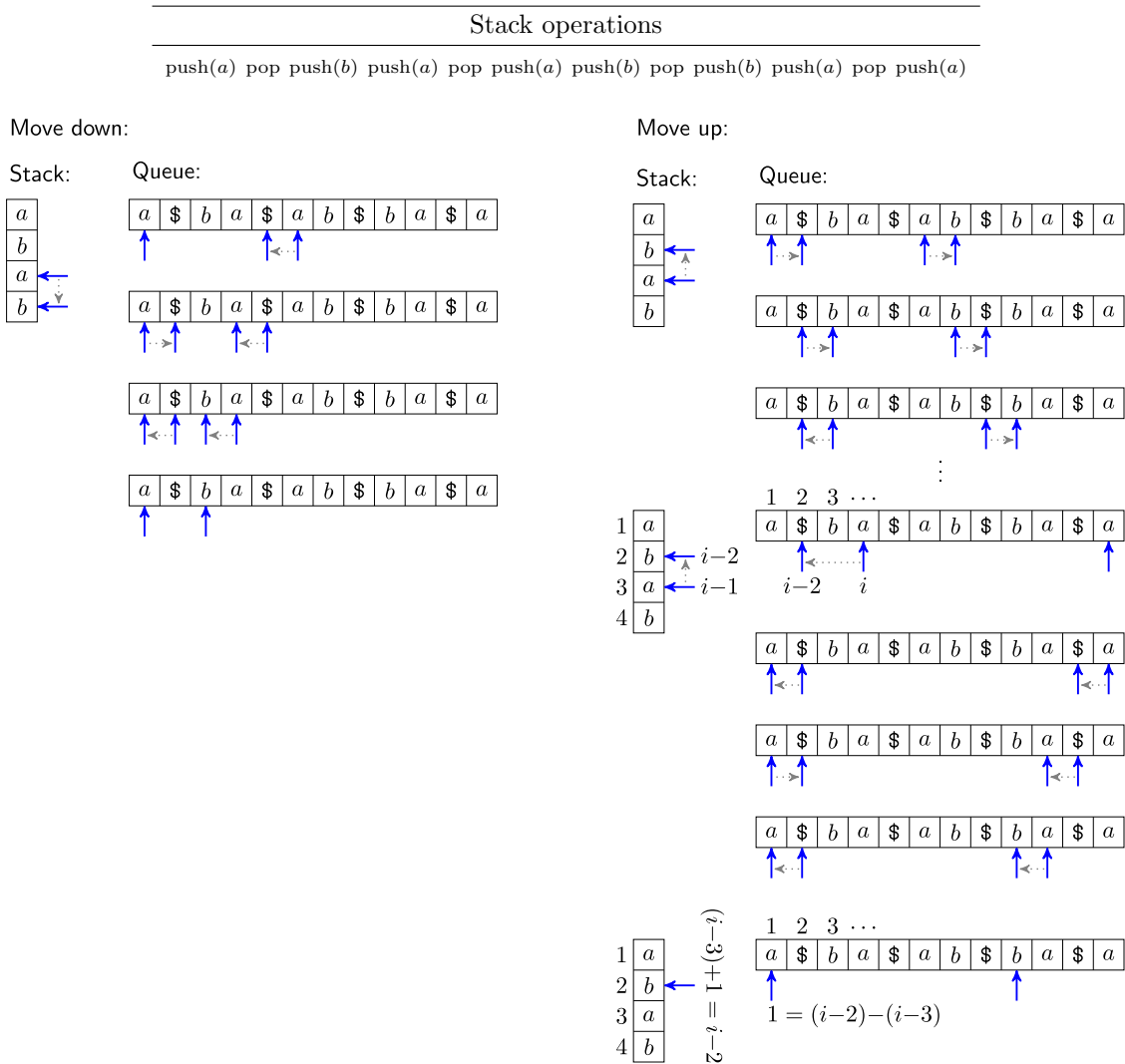


FIGURE 3. Example for Proposition 4.6. The stack operations on top of the figure induce the stack and queue contents depicted. Let the stack head be inside the stack pointing to the a next to the bottom of the stack. The corresponding symbol in the queue is the third a to which the right queue head is pointing. The left queue head is at the front of the queue. On the left part of the figure the simulation of a downward move of the stack head is depicted and on the right part the simulation of an upward move.

Proposition 3.4 shows that every finite-turn DIDQA accepting a unary language can be simulated by some DNEA. In fact, when the number of turns is reduced to zero, the $DIDQA_0$ can be simulated by some DNEA even for languages over arbitrary alphabets. Even better, the converse is also true.

Theorem 4.8. *The families of languages $\mathcal{L}(DNEA)$ and $\mathcal{L}(DIDQA_0)$ coincide.*

Proof. A D1DQA that performs no turns can never remove a symbol from the queue. So, in terms of stack automata it is non-erasing. Let the D1DQA₀ be tail diving. The mutual simulations of both devices are straightforward. The D1DQA₀ simulates the given DNESA step by step, whereby the end of the queue mimics the top of the stack and *vice versa*. Whenever the DNESA pushes a symbol, the D1DQA₀ enters the symbol into the queue and *vice versa*. Whenever the DNESA moves its stack head, the D1DQA₀ does the same with its queue head and *vice versa*. The DNESA detects when its stack head is at the bottom of the stack by reading the bottom-of-stack symbol, whereas the D1DQA₀ detects when its queue head has reached the opposite end of the queue by scanning a correspondingly marked symbol. \square

So, in particular for unary languages we have the inclusions

$$\mathcal{L}(\text{D1DQA}_k) \subseteq \mathcal{L}(\text{DNESA}) \subseteq \mathcal{L}(\text{D1DQA}_0),$$

for all $k \geq 1$. This implies $\mathcal{L}(\text{D1DQA}_k) = \mathcal{L}(\text{D1DQA}_0)$, and we derive that there is no turn hierarchy for unary D1DQA, *that is*, no turn is as good as any finite number of turns.

Corollary 4.9. *Let $k \geq 0$ be a constant. For any D1DQA_k accepting a unary language an equivalent D1DQA₀ can effectively be constructed.*

The construction in the proof of Theorem 4.8 shows the mutual simulations of DNESA and D1DQA₀. However, it reveals even more, the mutual simulations are possible step by step. So, any time constraint obeyed by the DNESA is obeyed by the D1DQA₀ and *vice versa*. In [20] it is shown how a weakly quasi-realtime DNESA accepts some encodings of the P-complete *monotone circuit value problem*. So, by Theorem 4.8 and Proposition 3.1 it follows that a P-complete language is accepted by some weakly realtime D1DQA₀.

Corollary 4.10. *The membership problem for the family of languages accepted by weakly realtime D1DQA₀ is P-hard.*

In order to derive an upper bound for the complexity of the membership problem, the question arises to what extent the conditions of working in weakly realtime or being turn bounded can be relaxed such that the problem is still in P. From Theorem 4.4 we already know that the condition of being turn bounded can be relaxed as long as the devices work in weakly realtime. In particular, this is even true for weakly quasi-realtime DDQA. The next theorem shows that the realtime condition can be relaxed for D1DQA as long as they are turn bounded.

Theorem 4.11. *Let $k \geq 0$ be a constant. The family of languages accepted by D1DQA_k is included in the complexity class P.*

Proof. Let $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \perp, F \rangle$ be a tail diving D1DQA_k, for $k \geq 0$, *that is*, the head at the front of the queue never moves inside the queue.

Recall that since M is k -turn bounded, the computation of M on some input is divided into at most $2k + 1$ phases. The first phase is an enqueueing phase that starts at initial time with an empty queue. It continues as long as M does not remove any symbol from the queue. When this happens, the next phase which is a dequeuing phase starts and continues as long as no symbol is entered into the queue. When this happens, the next enqueueing phase starts, and so on. After phase $2k$ the k turns are done and there might be a further final enqueueing phase.

First, we show that any halting computation of M takes at most a polynomial number of steps. To this end, assume that at the beginning of an enqueueing phase the length of the queue content is bounded by $c \cdot m$, where c is some constant and m denotes the number of input symbols read so far. This is certainly true at the beginning of a computation. We consider the behavior of M when the queue head is at the end of the queue and M is about to perform a sequence of λ -steps. Without moving the head into the queue, eventually the behavior can either be halting, reading a further input symbol, finishing the current enqueueing phase by removing a symbol from the front of the queue, or entering another symbol to the queue. Since M is halting by assumption, the

sequence of consecutive λ -steps until one of these cases occurs is bounded by a constant. Now assume that M moves its head into the queue. Then the behavior of M can entirely be described by a table that lists for every state M might be in and every pair of queue symbols scanned by the queue heads, what eventually happens. This can either be halting, reading a further input symbol, or moving the queue head to the end of the queue again. In the latter case, the table lists the state in which M returns the head. There are only finitely many of such tables. If M performs a sequence of λ -steps in the enqueueing phase without reading further input symbols, then it may move the queue head several times into the queue and back. However, since the table is fixed during such period, the number of consecutive λ -steps is bounded by $|Q| \cdot c \cdot m$ since M would run into a loop if it is in the same state with its queue head on the same position twice. Next, assume that M enters a further input symbol to the queue while performing the sequence of λ -steps. Then the table changes. But since there are only finitely many tables, again, M would run into a loop if it is in the same state with its queue head at the end of the queue and the same table twice. So, the sequence of λ -steps ends either by halting, reading a further input symbol, or finishing the current enqueueing phase. The number of such consecutive λ -steps is bounded by the constant $c_1 = |Q| \cdot c \cdot m \cdot t$, where t is the number of different tables that depends on Q and Γ only. We conclude that in an enqueueing phase at most c_1 steps are performed between reading two input symbols. Let n be the length of the input, then any enqueueing phase takes no more than $c_1 \cdot n \in O(n^2)$ steps. Moreover, the length of the queue increases by at most $|Q| \cdot t$ symbols between reading two input symbols and, thus, by at most $O(n)$ symbols in each enqueueing phase. That is, the length of the queue content is always linear in n .

Next, we consider dequeuing phases. In order to derive an upper bound for the number of steps in such a phase, assume M empties the queue entirely by λ -steps, *that is*, removes at most $O(n)$ symbols. In between removing two symbols, the queue head may again move into the queue. However, in order to avoid running into a loop, M may not be in the same state with the queue head on the same position twice. So, in between removing two symbols, M may perform at most $|Q| \cdot O(n) \in O(n)$ λ -steps. This implies that any dequeuing phase takes no more than $O(n^2)$ steps.

Since the number of phases is at most $2k + 1$ and k is a constant we conclude that any halting computation of M takes at most $O(n^2)$ steps.

Finally, M is simulated by a deterministic Turing machine. The simulation of halting computations certainly can be done in polynomial time. In order to cope with the problem of looping computations of M , the Turing machine maintains a counter that counts the number of consecutive λ -steps. If this number exceeds the bounds for halting computations, the Turing machine halts and rejects. Again the counter can be implemented with polynomial time. \square

Clearly, a further relaxation of conditions for D1DQA leads to a language family no longer included in \mathbf{P} , since even DQA are capable of universal computations. Dropping the realtime conditions for DDQA and imposing a turn bound at the same time seems to lead out of \mathbf{P} as well. Though we do not have a proof, the next example at least shows that a DDQA₀ can halt after performing an exponential number of steps on unary input.

Example 4.12. Let M be the DDQA₀ with unary input alphabet $\{a\}$, whose behavior is sketched as follows. Basically, M reads one input symbol and then runs through a phase of λ -steps. In this phase, it uses its two queue heads to simulate a deterministic two-way 2-head finite automaton. These devices can accept the unary language $L = \{a^{2^n} \mid n \geq 0\}$ (see, *e.g.*, [14]). In detail, M enters a symbol to the queue and, subsequently simulates a deterministic two-way 2-head finite automaton accepting L . If the length of the queue is a power of two (and, thus, the simulation ends accepting), M finishes the phase. Otherwise, it enters another symbol to the queue and simulates the 2-head finite automaton again, and so on until the queue length is a power of two. Then the behavior is repeated with the next input symbol. So, when M halts since the input a^n has been read entirely, the length of the queue is 2^n . Hence, M performs an exponential number of steps before halting.

Theorem 4.4, Corollary 4.10, and Theorem 4.11 imply the \mathbf{P} -completeness of the membership problems for the families of languages accepted by (weakly realtime) D1DQA _{k} as well as for the family of languages accepted by weakly realtime DDQA.

Corollary 4.13. *Let $k \geq 0$ be a constant. The membership problems for the families of languages accepted by (weakly realtime) $D1DQA_k$, as well as the family of languages accepted by weakly realtime $DDQA$ are P -complete.*

5. DIVING AND DECIDABILITY

It is shown in [16] that the commonly studied decidability questions such as emptiness, finiteness, universality, inclusion, and equivalence are not semidecidable for DQA , whereas the questions of emptiness and finiteness are decidable for finite-turn DQA . Since the language family accepted by DQA (with an unbounded number of turns) is contained in those accepted by $D1DQA$ and $DDQA$, we obtain from the first result that all above-mentioned decidability questions are not semidecidable for $D1DQA$ and $DDQA$ as well. Thus, we will investigate in this section the decidability status for the remaining models, namely weakly realtime $DDQA_k$ and weakly realtime $D1DQA_k$, and we will show that all above-mentioned decidability questions are not semidecidable for weakly realtime $DDQA_k$, whereas it turns out that the problems of emptiness and finiteness are decidable for several subclasses of $D1DQA_k$.

Let us start with the non-semidecidability results for weakly realtime $DDQA_k$. A widely used tool to prove such non-semidecidability results is to use the set $VALC(T)$ of *valid computations* of a Turing machine T (see, e.g., [10]) which are basically histories of accepting Turing machine computations and allow to translate non-semidecidability results obtained for Turing machines to non-semidecidability results of the models under consideration. This method has been applied, for example, in [16] to obtain the non-semidecidability of the above-mentioned questions for DQA . Thus, we now show that the set $VALC(T)$ can be accepted by some weakly realtime $DDQA_0$ which implies non-semidecidability results for weakly realtime $DDQA_0$ as well.

Lemma 5.1. *Let T be a Turing machine. Then, a weakly realtime $DDQA_0$ accepting $VALC(T)$ can effectively be constructed.*

Proof. It is shown in [12] how a two-head deterministic finite automaton M with one-way input tape accepting $VALC(T)$ can effectively be constructed. Now, a $DDQA_0$ M' can accept the set $VALC(T)$ as follows. First, the input is stored in the queue. Second, the 2-head automaton M is simulated by M' with the help of its two queue heads. Finally, M' accepts if M accepts and rejects otherwise. Clearly, M' performs no turn and thus is a weakly realtime $DDQA_0$. \square

This lemma yields the following non-semidecidability results.

Theorem 5.2. *Let $k \geq 0$ be a constant and M be a weakly realtime $DDQA_k$. Then the questions of emptiness, finiteness, universality, inclusion, and equivalence are not semidecidable for M . Moreover, it is not semidecidable whether or not a given weakly realtime $DDQA$ is a $DDQA_k$.*

Proof. Since $VALC(T)$ is empty or finite for some Turing machine T if and only if $L(T)$ is empty or finite, we obtain that emptiness and finiteness are not semidecidable. Testing inclusion in the empty set or equivalence with the empty set gives that inclusion and equivalence are not semidecidable as well. Finally, we consider the set $INVALC(T)$ of invalid computations of T which is the complement of $VALC(T)$ with respect to a suitable encoding alphabet. Since two-head deterministic finite automata with one-way input tape are closed under complementation, we can construct similar to Lemma 5.1 a weakly realtime $DDQA_0$ accepting $INVALC(T)$. Since testing the universality of $INVALC(T)$ is equivalent to test the emptiness of $VALC(T)$, we yield that universality is not semidecidable as well.

For a given weakly realtime $DDQA_k$ N with input alphabet Σ where $\$ \notin \Sigma$, we construct a $DDQA$ N' with input alphabet $\Sigma \cup \{\$\}$. As long as no $\$$ appears in the input N' just simulates N . But when N' reads the first $\$$ two cases are considered: If N' is in a non-accepting state, then it just halts. If N' is in an accepting state, then it performs $k + 1$ turns and then halts. Clearly, N' is a weakly (quasi)-realtime $DDQA_k$ if and only if $L(N)$ is empty. If we could semidecide whether N' is a $DDQA_k$, we could also semidecide whether $L(N)$ is empty which is a contradiction. Thus, the second claim follows. \square

Next, we are going to investigate the decidability questions for the class of finite-turn D1DQA. As first result we can note that the inclusion problem is not semidecidable for weakly realtime D1DQA_k with $k \geq 1$, since it is not semidecidable for realtime DQA_k owing to the results shown in [16]. It remains to consider the case $k = 0$. To this end, we first show the following result saying that a D1DQA can always simulate a DQA by saving one turn.

Lemma 5.3. *Let $k \geq 1$ be a constant and M be realtime DQA_k. Then an equivalent weakly realtime D1DQA_{k-1} M' can effectively be constructed.*

Proof. We recall that a DQA_k M processes its input in $2k + 1$ phases alternating between enqueueing and dequeuing phases, where the $(2k)$ th phase is a dequeuing phase and the $(2k + 1)$ st phase is an enqueueing phase. We construct an equivalent D1DQA_{k-1} M' that simulates the first $2k - 1$ phases in the same way as in M , where M' counts by using its state set which phase is currently simulated. We note that M' has made at most $k - 1$ turns after having simulated phase $2k - 1$. To simulate the remaining two phases no further turn is allowed. For the dequeuing phase $2k$ the tail head moves by λ -steps to the leftmost symbol of the queue and moves for every dequeued symbol in M one position to the right. When the final phase $2k + 1$ has to start, M' stores the symbol that the tail head currently scans in its state set, since this is the leftmost symbol of M' 's queue. Subsequently, the tail head moves by λ -steps to the rightmost symbol of the queue and starts to simulate phase $2k + 1$ in the same way as in M . We observe that M' is equivalent to M , works in weakly realtime, and performs at most $k - 1$ turns. \square

Using this lemma we obtain that the inclusion problem for DQA₁ can be reduced to the inclusion problem for D1DQA₀ which implies the non-semidecidability of the latter problem.

Theorem 5.4. *Let $k \geq 0$ be a constant and M, M' be two weakly realtime D1DQA_k. Then it is not semidecidable whether or not $L(M) \subseteq L(M')$.*

For the questions of emptiness and finiteness we will now obtain positive decidability results, since it turns out to be possible to simulate subclasses of finite-turn D1DQA by nondeterministic stack automata for which it is known that emptiness and finiteness are decidable. It is currently an open problem whether these positive results can be extended to the whole class of finite-turn D1DQA.

It has been shown in Proposition 3.4 that D1DQA_k accepting unary languages can be simulated by non-erasing stack automata. Thus, we immediately obtain the decidability of emptiness and finiteness for such D1DQA_k, since it is shown in [8, 23] that both questions are decidable for nondeterministic stack automata.

We will now reconsider the classes (∞, ℓ) -D1DQA_k and (ℓ, ∞) -D1DQA_k defined in Section 4. We will show in the sequel that both classes of D1DQA_k can be simulated by nondeterministic stack automata and hence share their decidable emptiness and finiteness problems.

Proposition 5.5. *Let $k, \ell \geq 0$ be constants. Then the family $\mathcal{L}((\infty, \ell)\text{-D1DQA}_k)$ is included in $\mathcal{L}(\text{NNESA})$.*

Proof. The case $k = 0$ has been shown in Theorem 4.8. So, let $k \geq 1$ and M be a (∞, ℓ) -D1DQA_k, and assume that M has a queue head at the end of the queue. Then M performs k phases in which symbols are removed from the queue. Moreover, in every dequeuing phase at most ℓ queue head reversals take place. The principal idea of the simulation is as in the proof of Theorem 4.8, but we have to cope with the fact that an NNESA is not able to remove symbols from the top of the pushdown store and, in particular, not from the bottom of the pushdown store which is the front of the simulated queue. Hence, the basic idea to simulate the removal of symbols from the bottom of the pushdown store, is to decide nondeterministically at the moment when the symbol is pushed onto the pushdown store, *that is*, when it is enqueueing, in which of the k dequeuing phases it will be removed and how many of the ℓ queue head reversals have been performed up to that moment in this phase. The nondeterministic choice is written down as double index from $\{1, 2, \dots, k\} \times \{1, 2, \dots, \ell\}$ to the symbol pushed. If it is guessed that the symbol will never be removed, the symbol pushed has no index. By using its state set M' can check that the symbols pushed start with index $(1, 1)$ and are numbered in ascending and consecutive order in the first component and in ascending order in the second component of the double

index. Additionally, it is ensured that once a symbol with no index has been pushed all further symbols pushed are not indexed as well.

Now, M' simulates M as follows. In its state set the simulated state of M and the symbol at the front of the queue are stored. In each enqueueing phase of M the symbols with guessed index are pushed onto the pushdown store where the indices obey the above-described order. The moves of M inside the queue can be simulated by M' with suitable moves inside the pushdown store. To simulate the m th dequeuing phase in which m' queue head reversals have been performed so far, which can be counted by M' in its state set, M' enters the stack and searches the bottom-most stack symbol indexed by (m, m') . Subsequently, M' checks whether all stack symbols indexed by (m, m') would have been removed by M , M' updates the symbol at the front of the queue stored in its state set, and returns to the top of the stack, where the simulation of M is continued. M' accepts its input when all phases have successfully been completed and M would halt in an accepting state. \square

Corollary 5.6. *Let $k, \ell \geq 0$ be constants and M be a (∞, ℓ) -D1DQA $_k$. Then the emptiness and finiteness of $L(M)$ is decidable.*

A similar simulation result can be obtained for $\mathcal{L}((\ell, \infty)$ -D1DQA $_k$). However, the property of the simulating stack automaton to be non-erasing gets lost.

Proposition 5.7. *Let $k, \ell \geq 0$ be constants. Then the family $\mathcal{L}((\ell, \infty)$ -D1DQA $_k$) is included in $\mathcal{L}(\text{NSA})$.*

Proof. Let M be an (ℓ, ∞) -D1DQA $_k$ and assume again that M has a queue head at the end of the queue. Let us first consider the case when $k \leq 1$ and M has only one enqueueing phase which is divided into $n + 1 \leq \ell + 1$ *checking* subphases in which the queue head is inside the queue and has already been n times inside the queue and at most $n + 1$ *enqueueing* subphases in which symbols are enqueued. Both types of subphases are alternating.

The construction of an NSA M' simulating M can be sketched as follows. At first, M' guesses at the beginning of the computation the complete contents of the queue that will be enqueued by M and stores the symbols in reversed order in the pushdown store. Moreover, by adding a suitable index to each symbol pushed it is guessed in which subphase the symbols are enqueued. That is, the symbols enqueued in the last subphase are at the bottom of the pushdown store whereas the first enqueued symbols are at the top of the pushdown store. Second, M' has to simulate the enqueueing phase of M : Each enqueueing subphase is simulated by reading the input and checking the correct guessing of the queue contents by moving inside the pushdown store. Note that the correct position inside the pushdown store can be identified by the index of the symbols pushed. Each checking subphase is simulated by reading the input and moving suitably inside the pushdown store. Here the simulation starts from the bottom of the pushdown store which is the end of queue. Finally, M' can simulate the dequeuing phase of M as follows. Since the queue contents are written down in reversed order, each dequeuing in M can be simulated by popping a symbol from the pushdown store in M' . Each checking subphase in the dequeuing phase is simulated as in the enqueueing phase.

This construction is straightforwardly generalized to $k \geq 1$ turns and $k + 1$ enqueueing phases. At the beginning of the computation M' guesses the complete contents of the queue in reversed order in its pushdown store and adds to each symbol a double index indicating in which phase and subphase it will be enqueued. Each enqueueing and dequeuing phase can then be simulated as above. The only difference is that the end of the queue does not start from the bottom of the pushdown store, but starts from the bottom-most pushdown symbol indexed by the current simulated phase. \square

Corollary 5.8. *Let $k, \ell \geq 0$ be constants and M be an (ℓ, ∞) -D1DQA $_k$. Then emptiness and finiteness of $L(M)$ are decidable.*

Finally, we obtain that the two restrictions of D1DQA $_k$ which lead to decidable emptiness and finiteness problems are themselves decidable for D1DQA $_k$.

Theorem 5.9. *Let $k, \ell \geq 0$ be constants and M be a D1DQA $_k$. Then it is decidable whether or not (a) M is a (∞, ℓ) -D1DQA $_k$ and (b) M is an (ℓ, ∞) -D1DQA $_k$.*

Proof. For a given D1DQA_k M and some fixed $\ell \geq 0$, we construct a D1DQA_k M' that simulates M , but accepts if and only if M tries to perform more than ℓ queue head reversals in some dequeuing phase (regardless of whether the computation of M is accepting or rejecting).

This can be realized by simulating M while counting in an additional component of its state set the number of queue head reversals executed in the current phase so far. M' accepts an input as soon as it would require more than ℓ queue head reversals in one phase. Otherwise, the input is rejected. Clearly, M' is a (∞, ℓ) - D1DQA_k and accepts the empty set if and only if M is a (∞, ℓ) - D1DQA_k as well. Now, claim (a) follows from the decidability of emptiness for (∞, ℓ) - D1DQA_k shown in Corollary 5.6. Claim (b) can be shown similarly. \square

REFERENCES

- [1] A.V. Aho, Indexed grammars – an extension of context-free grammars. *J. ACM* **15** (1968) 647–671.
- [2] A.V. Aho, Nested stack automata. *J. ACM* **16** (1969) 383–406.
- [3] F.-J. Brandenburg, On the intersection of stacks and queues. *Theor. Comput. Sci.* **58** (1988) 69–80.
- [4] G. Buntrock and F. Otto, Growing context-sensitive languages and Church-Rosser languages. *Inform. Comput.* **141** (1998) 1–36.
- [5] A. Cherubini, C. Citrini, S. Crespi-Reghizzi and D. Mandrioli, QRT FIFO automata, breadth-first grammars and their relations. *Theor. Comput. Sci.* **85** (1991) 171–203.
- [6] N. Chomsky, Context-free grammars and pushdown storage. Vol. 65 of Quarterly Progress Report. MIT Research Laboratory of Electronics, Massachusetts (1962) 187–194.
- [7] R.H. Gilman, A shrinking lemma for indexed languages. *Theor. Comput. Sci.* **163** (1996) 277–281.
- [8] S. Ginsburg, S.A. Greibach and M.A. Harrison, One-way stack automata. *J. ACM* **14** (1967) 389–418.
- [9] S. Ginsburg, S.A. Greibach and M.A. Harrison, Stack automata and compiling. *J. ACM* **14** (1967) 172–201.
- [10] J. Hartmanis, Context-free languages and Turing machine computations, in vol. of 19 *Proc. Symposia in Applied Mathematics* (1967) 42–51.
- [11] J. Hartmanis, On non-determinacy in simple computing devices. *Acta Inform.* **1** (1972) 336–344.
- [12] M. Holzer, M. Kutrib and A. Malcher, Complexity of multi-head finite automata: origins and directions. *Theor. Comput. Sci.* **412** (2011) 83–96.
- [13] J.E. Hopcroft and J.D. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA (1979).
- [14] M. Kutrib, On the descriptive power of heads, counters, and pebbles. *Theor. Comput. Sci.* **330** (2005) 311–324.
- [15] M. Kutrib and A. Malcher, Reversible pushdown automata. *J. Comput. System Sci.* **78** (2012) 1814–1827.
- [16] M. Kutrib, A. Malcher, C. Mereghetti, B. Palano and M. Wendlandt, Deterministic input-driven queue automata: finite turns, decidability, and closure properties. *Theor. Comput. Sci.* **578** (2015) 58–71.
- [17] M. Kutrib, A. Malcher and M. Wendlandt, Input-driven queue automata with internal transductions, in *Language and Automata Theory and Applications (LATA 2016)*. Vol. 9618 of *Lect. Notes Comput. Sci.* Springer (2016) 156–167.
- [18] M. Kutrib, A. Malcher and M. Wendlandt, Reversible queue automata. *Fund. Inform.* **148** (2016) 341–368.
- [19] M. Kutrib, A. Malcher and M. Wendlandt, Queue automata: foundations and developments, in *Reversibility and Universality*. Vol. 30 of *Emergence, Complexity and Computation*. Springer (2018) 385–431.
- [20] K. Lange, A note on the P-completeness of deterministic one-way stack language. *J. UCS* **16** (2010) 795–799.
- [21] R. McNaughton, P. Narendran and F. Otto, Church-Rosser Thue systems and formal languages. *J. ACM* **35** (1988) 324–344.
- [22] B. Monien, Two-way multihead automata over a one-letter alphabet. *RAIRO: ITA* **14** (1980) 67–82.
- [23] W.F. Ogden, Intercalation theorems for stack languages. In *Symposium on Theory of Computing (STOC 1969)*. ACM Press (1969) 31–42.
- [24] R. Vollmar, Über einen Automaten mit Pufferspeicherung. *Computing* **5** (1970) 57–70.