Electronic Colloquium on Computational Complexity, Report No. 28 (2004)

Aggregates with Component Size One Characterize Polynomial Space

Arfst Nickelsen and Till Tantau
Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
10623 Berlin, Germany
{nicke,tantau}@cs.tu-berlin.de

Lorenz Weizsäcker
Humboldt Universität Berlin
Institut für Informatik
12489 Berlin, Germany
weizack@informatik.hu-berlin.de

March 19, 2004

Abstract

Aggregates are a computational model similar to circuits, but the underlying graph is not necessarily acyclic. Logspace-uniform polynomial-size aggregates decide exactly the languages in PSPACE; without uniformity condition they decide the languages in PSPACE/poly. As a measure of similarity to boolean circuits we introduce the parameter *component size*. We prove that already aggregates of component size 1 are powerful enough to capture polynomial space. The only type of cyclic components needed to make polynomial-size circuits as powerful as polynomial-size aggregates are binary xor-gates whose output is fed back to the gate as one of the inputs.

1 Introduction

Aggregates are a computational model which was introduced by Dymond and Cook in 1980 [DC80]. As in the circuit model, an aggregate consists of gates, each of them computing a binary boolean function. In order to perform more complex computations, the gates are connected to each other, forming a graph with the gates as nodes. The input is fed to the circuit through special input gates; similarly, the output is taken from special output gates. In contrast to circuits, the graph of an aggregate may contain cycles and a gate can be used several times during a computation. This entails a more sophisticated concept of evaluation: rather than assigning just one value to each gate during the computation, we assign a (possibly varying) value to each gate of an aggregate in each step of the computation.

There is an apparent gap between the computational power of polynomial-size circuits and polynomial-size aggregates: it is known that aggregate size corresponds to Turing-machine space [DC89, Hon86]. In particular, any language in PSPACE can be decided by a logspace-uniform polynomial-size family of aggregates. Proofs for this result up to now make extensive use of the possibility of connecting the gates in cycles.

We consider aggregates where the *component size*, i.e., the maximum number of nodes that are reachable from each other, is bounded. This parameter is important since it measures how much an aggregate does look like a circuit. The underlying graph of an aggregate with component size c is acyclic except for clusters of at most c gates. We show that restricting the component size does not reduce the computational power of aggregates. Even if we allow no cycles at all except loops on xor-gates, all languages in PSPACE can be decided.

ISSN 1433-8092

This paper is organized as follows. In Section 2 we review the aggregate model of Hong [Hon86]. We introduce uniform complexity classes USIZE(s), where the number of gates in the aggregate is bounded by a size bound s, and show that these classes are equal to DSPACE(s) for well-behaved space bounds s. We also show that the nonuniform classes SIZE(s) correspond to Turing machine space classes with advice linear in s. In Section 3 we introduce and explain some special gadgets for aggregates that we will use later on. The most simple one is the *flipflop*, which is obtained by connecting the outgoing edge of an xor-gate to one of its ingoing edges. We prove our Main Theorem in Section 4 and also present a nonuniform version of it. In Section 5 we point out the connection between our results and techniques of Section 4 and the theory of leaf language characterisations of complexity classes. We explain how to obtain a weaker version of the Main Theorem using the fact that PSPACE has a regular leaf language.

2 Basics on aggregates

In this section we give a definition of aggregates, discuss uniformity conditions, define parameters like size and computation time of aggregates, and introduce aggregate language classes. We will use the definition of aggregates of Hong [Hon86] which differs from the definition of Dymond and Cook [DC80, DC89] regarding the input scheme. Their input scheme is more elaborate and allows aggregates of sublinear size, but we do not consider such size bounds.

2.1 Description of the Computational Model

Aggregates can be regarded as circuits where the condition that the underlying graph has to be acyclic simply is dropped. However, this does not immediately yield a computational model since the absence of cycles is crucial in the circuit model, since the canonical definition of the function computed by a given circuit depends on this property: First, a number depth(v) is assigned to each gate v in the circuit's graph, where depth(v) is defined as the length of longest path from a node with zero fan-in to v. Then recursively a value in $\{0,1\}$ is assigned to all gates. Gates of depth zero get a value depending on their labeling or their corresponding input bit. Gates of greater depth are assigned a value depending on the gate function and the values of its parent gates.

This procedure is obviously not applicable to graphs containing cycles. Although one can still assign values to gates with zero fan-in, gates can be connected in such a way that neither of them can be assigned a value before the other one has been assigned a value. This problem is solved by assigning an initial value, for instance 0, to all non-constant gates and by then starting a stepwise re-evaluation according to each gate's function and its parents' values. Since there is no reason why after any number of re-evaluations the values should stop to change, we stop the stepwise re-evaluation as soon as a special "flag gate" outputs 1.

2.2 Formal Definition of the Model

We introduce the following notions: aggregates, computation of an aggregate, function computed by an aggregate.

An *aggregate* D is a directed graph with labeled nodes and labeled edges. Every node v has in-degree 0, 1, or 2, and a node with in-degree i is labeled with a function $f_v\colon \{0,1\}^i \to \{0,1\}$. Its incoming edges have distinct labels in $\{1,2\}$ such that for non-symmetric functions f_v the first and the second argument can be distinguished. The input bits $x_1, \ldots, x_n \in \{0,1\}$, where n is the *input length* of D, are delivered to the aggregate via n special *input gates*. These are gates of

in-degree zero and are labeled with a number $i \in \{1, \ldots, n\}$. The specification of D is completed by indicating the *output gates* and the *flag gate*. The output gates are indicated by an m-tuple whose ith entry tells us which gate yields the ith output bit. The number of gates in an aggregate is referred to as its *size*.

For an aggregate D let V be the set of its nodes and n its input length. Together with an input $x = x_1 \cdots x_n \in \{0,1\}^n$ it induces the *value function* $\mathrm{val}_{D,x} \colon V \times \mathbb{N} \to \{0,1\}$, which assigns a boolean value to every node v in every *step* t. For $t = 0$ we set $\mathrm{val}_{D,x}(v,0) = 1$ for every node v with in-degree zero that is labeled 1 and for input gates with a label i such that $x_i = 1$. We set $\mathrm{val}_{D,x}(v,0) = 0$ for all other nodes. For every step $t > 0$ the value $\mathrm{val}_{D,x}(v,t)$ is defined according to v's label and the values of v's parent nodes w_1 and w_2 at step $t-1$, where the edge from w_i to v has label i:

$$\mathrm{val}_{D,x}(v,t) := f_v\big(\mathrm{val}_{D,x}(w_1,t-1), \mathrm{val}_{D,x}(w_2,t-1)\big).$$

This re-evaluation is defined analogously when v's in-degree is one.

The *length of the computation* or the *computation time*, denoted t_{stop}, is the least $t \in \mathbb{N}$ for which the flag gate has value 1. The *output of D on input x* are the values of the output gates at step t_{stop}. If the flag gate has value 0 in every step $t \in \mathbb{N}$, neither an output nor the computation time is defined. The *function computed by D* is a partial boolean function from $\{0,1\}^n$ to $\{0,1\}^m$, where n is the number of input gates and m is the number of output gates. This function need not be total since the computation may deliver no output. A function $f \colon \{0,1\}^* \to \{0,1\}^*$ is *computed by a family of aggregates* $(D_n)_{n\in\mathbb{N}}$ if for every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$ the aggregate D_n computes $f(x)$ upon input x.

The size of an aggregate with n input gates is at least n. In contrast, the definition of aggregates given in [DC89] uses a special input scheme that allows aggregates with sublinear size.

Just as for circuits, complexity classes can be defined via aggregates by asymptotically restricting the size and/or the computation time that the families of aggregates are allowed to use in order to compute a function on $\{0,1\}^*$. In this paper we consider only size classes. Before introducing such classes, we fix a notion of uniformity for this paper, namely logspace uniformity.

Definition 2.1. A family of aggregates $(D_n)_{n\in\mathbb{N}}$ is *logspace-uniform*, if there exists a Turing machine that outputs (a standard encoding of) D_n for every input of length n using $O\big(\log s(n)\big)$ space, where $s(n)$ is the size of D_n.

In this definition $s(n)$ can be replaced by the length of the encoding, since, although the encoding length is not exactly specified, in any case the two values are polynomially related.

2.3 Size Classes

Definition 2.2. A language $A \subseteq \{0,1\}^*$ is in $\mathrm{USIZE}(s)$ if there exists a logspace-uniform family of aggregates $(D_n)_{n\in\mathbb{N}}$ that computes the characteristic function χ_A of A, where the size of each D_n is bounded by $O(s)$.

It is known [DC89, Hon86] that the size of uniform aggregates corresponds to Turing machine space. Of course, for the aggregate model presented here this only holds for bounds with at least linear growth. Indeed, we need another assumption on the bounds to make this correspondence hold, see the following definition and theorem.

Definition 2.3. A function $f \colon \mathbb{N} \to \mathbb{N}$ is a *proper size function* if $f \in \Omega(n)$ and there exist a Turing machine M_f that on every input of length n outputs $1^{f(n)}$, using at most $O\big(\log f(n)\big)$ space.

The proof of the following theorem is similar to the well-known proof that a language A is in P if and only if its decidable by logspace-uniform polynomial size circuits [Sav72]. The details can be taken from that proof, see for example [BDG95].

Theorem 2.4. $\mathrm{USIZE}(s) = \mathrm{DSPACE}(s)$ *for proper size functions* s.

Proof. Let $A \in \mathrm{USIZE}(s)$ via an aggregate family $(D_n)_{n\in\mathbb{N}}$. The computation of D_n for a given input x of length n can be simulated by a Turing machine M. First, M constructs the encoding of D_n and then simulates the computation. Besides the encoding, only the values of the gates have to be stored. Since the family $(D_n)_{n\in\mathbb{N}}$ is logspace-uniform, the construction of D_n does not affect the $O(s)$ space bound of M.

For the other direction, let M be a Turing machine with an $O(s)$ space bound. Any configuration of M on an input of size n can be mapped to the next configuration by a layered circuit of size $g(n)$, $g \in O(s)$, which has constant depth. It can be built by a Turing machine that needs storage only to compute the correct size $g(n)$. The circuit is fed with its own output so that in step $i \cdot d$ it yields the ith configuration of M's computation, where d is the circuit's depth.

The same output is also given to a second circuit of logarithmic depth that checks for each incoming configuration whether a final state has been reached. If so, it makes the aggregate output the result. Inputs that do not represent a configuration of M are ignored. Again, the only space needed for the construction is that for the computation of the right size. Both the sizes of the first and second circuits can be assumed to be proper size functions since s is a proper size function. Hence the construction of the aggregate that computes the characteristic function of $\mathrm{L}(M)$ is logspace-uniform and of size $O(s)$. □

Non-uniform variants of the size classes have a simple characterization as well. The definition below differs from Definition 2.2 only insofar that the logspace-uniformity condition is omitted.

Definition 2.5. A language $A \subseteq \{0,1\}^*$ is in $\mathrm{SIZE}(s)$ if there exists a family of aggregates $(D_n)_{n\in\mathbb{N}}$ that computes the characteristic function χ_A of A, where the size of D_n is bounded by $O(s)$.

Just as the class of languages decided by nonuniform polynomial-size circuit families equals the advice class P/poly, nonuniform polynomial-size aggregates characterize PSPACE/poly. The following theorem formulates this relationship in a general way. For the definition of advice classes $\mathcal{C}/\mathcal{F}$ see [KL80].

Theorem 2.6. $\mathrm{SIZE}(s) = \mathrm{DSPACE}(s)/O(s)$ *for every* $s \in \Omega(n)$.

Proof. We follow the same procedure as in the proof of Theorem 2.4. To show that $\mathrm{SIZE}(s) \subseteq \mathrm{DSPACE}(s)/O(s)$, the Turing machine gets the encoding of D_n as advice. The length of the encoding is $O(s)$ since the gates have bounded fan-in.

For the opposite direction, where the aggregates D_n have to simulate the Turing machine, the advice for input length n can be fed to D_n via $O(s)$ constant gates. □

3 Components for Assembling Aggregates

In this section we present special gadgets, called components, like flipflops, bit counters, and bit enumerators that will be used in later proofs. A *component of an aggregate* is a part of an aggregate that has incoming and outgoing edges through which it gets (respectively delivers) input (respectively output) tuples throughout the computation.

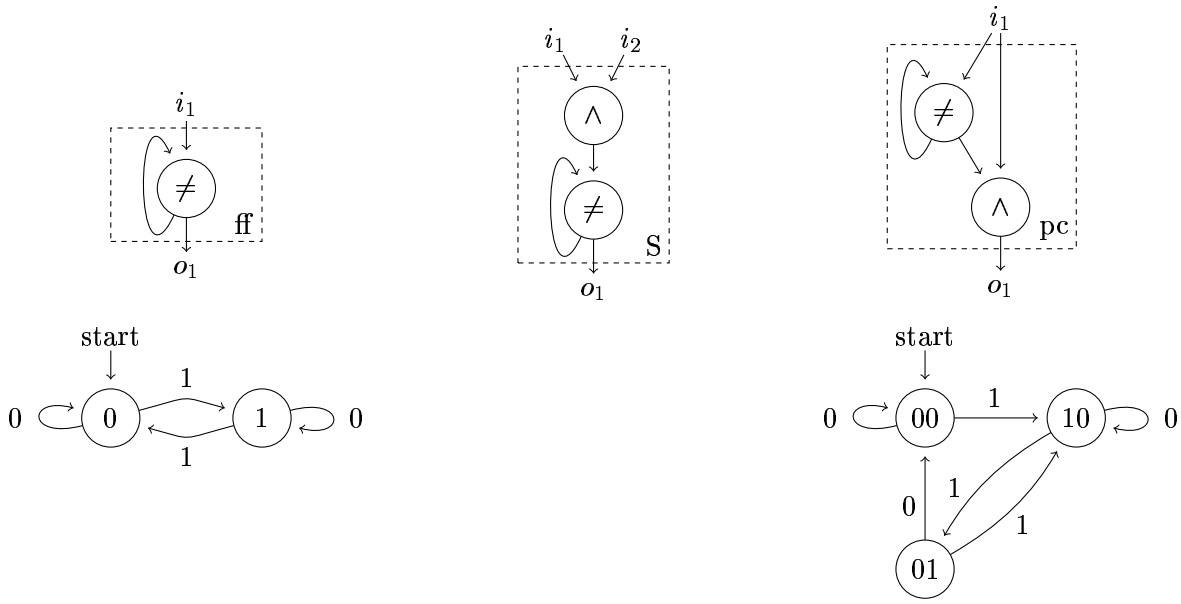

Figure 1: The flipflop, storage, and parity counter components. For the flipflop and for the parity counter, also the corresponding transition graphs are depicted. The marking of each state in the flipflop's transition graph equals the value of the xor-gate (= the output of the flipflop). The marking $s_1 s_2$ of a state in the transition graph of the parity counter means that the value of the xor-gate is s_1 and the value of the and-gate (= the output of the counter) is s_2.

A component has not only incoming and outgoing edges, but also internal edges. Therefore the output depends on both, the external input and the input from its own gates. The last can be seen as an internal state; thus components are much like (deterministic) finite state transducers. Such transducers start from an initial state reading one character per step from an infinite tape, passing to a next state according to their transition function and producing an output that is a function of the new state. In our case, the set of characters and the set of states are $\{0,1\}^k$ respectively $\{0,1\}^s$, where k is the number of incoming edges and s the number of gates of the component. The output is a selection of l bits from the s state bits, where l is the number of outgoing edges.

Flipflops. The first component shown in Figure 1 has just one incoming and one outgoing edge. Its only gate is an xor-gate, of which one input is fed by its output. The component changes its state, i.e., its output, after every incoming 1. Therefore we call it a *flipflop*.

Storage Components. The *storage component* S is a "controlled" flipflop. The first input line is used to feed an input signal to the flipflop. The second input line is the control line. Only when this control line is set to 1, the first input reaches the flipflop.

Parity Counters. A parity counter, see Figure 1 once more, is a component with one incoming and one outgoing edge. It outputs a 1 for every two 1's in the input stream. It can be built using an and-gate, where one input is fed directly by the input of the component while the second passes through a flipflop before it is fed to the and-gate.

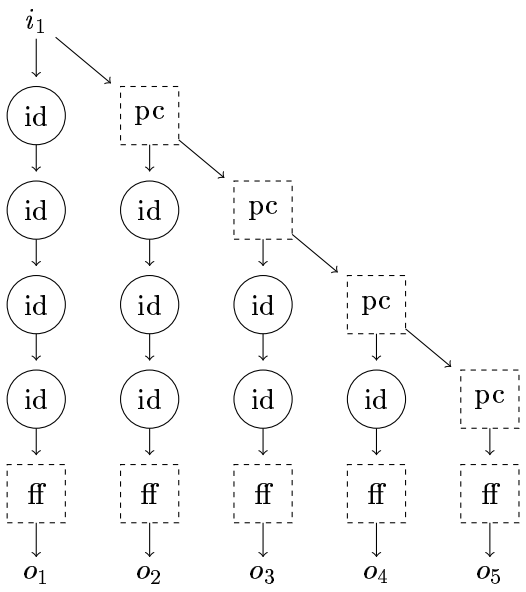

Figure 2: A 5-bit counter. The small boxes are components. Those with label "pc" are parity counters, while flipflops are labeled "ff."

Bit Counters. An *m-bit counter*, see Figure 2, outputs in binary the number of 1's which have been fed into its input so far modulo 2^m. The output is delivered with a certain delay. The component is constructed as follows: We connect $m-1$ parity counters into a chain. The kth parity counter outputs, with a delay of k steps, a single 1 for every 2^k 1's fed into the chain. The outputs of the parity counters are passed to flipflops that yield the kth bit of the counter. This bit has to flip every 2^k incoming 1's. By delaying the signal by $m-k$ steps, a constant delay of m steps can be attained. Thus the m flipflops deliver the number of incoming 1's in binary with a delay of m steps.

Bit Enumerators. An m-bit counter can easily converted to an *m-bit enumerator* that enumerates all bitstrings of length m without getting any input. This is done by feeding the m-bit counter with a 1-gate. The enumeration starts in step m with 0^m, which is also the output until that step. As the counter's input is always fed with a 1, the output is incremented in each step. When all words of length m have been enumerated, the enumeration starts over again without any delay.

Deterministic Finite Automata. In general, a deterministic finite automaton M can be simulated by a component B as follows. Let M's input symbols be encoded using k bits. The component B has k incoming edges. Additionally, there is one incoming edge that should be fed with a 1 only in those step in which B should interpret the input on the other edges as an input symbol for M. The signal must not necessarily be present at regular intervals, but it often is convenient that it is present every d steps for some d that depends on the construction of the aggregate. The component has a single outgoing edge. A 1 on that edge indicates that, with a delay of d steps, the automaton M accepts the input fed in so far.

The component B uses a subcomponent E to store a coding of the current state of M. When an input symbol is delivered, a layered circuit C is used that maps that symbol and the current

state to the new current state according to M's transition function. The result is stored in E. The circuit C also computes a bit that indicates whether the new current state is accepting or not. This bit is stored in E as well.

4 Characterization of Polynomial Space by Component Size One

We have seen in Section 2 that uniform polynomial-size aggregate decide exactly those languages that are in PSPACE. Is this still true if we impose restrictions on the graphs of the aggregates? We are especially interested in restrictions that in some sense make aggregates more similar to circuits. One way to do so is to bound the *component size* of aggregates, i. e., the maximal number of gates that are mutually reachable from each other. Surprisingly, even component size one is sufficient to capture PSPACE. Note that a "smaller" component size than one yields essentially circuits again, which accept the class P. Thus component size one is optimal.

4.1 Loops on Xor-Gates

We now show that PSPACE is characterized by aggregates with component size one. In fact, it suffices to consider only aggregates whose graphs are acyclic except for loops in flipflops.

Theorem 4.1 (Main Theorem). *A language $A \subseteq \{0,1\}^*$ is recognized by a family of logspace-uniform polynomial size aggregates that are acyclic except for loops on xor-gates iff $A \in$ PSPACE.*

Proof. For the only-if part, note that restrictions on the graph of the aggregates do not affect the fact that their computations can be simulated by Turing machines using polynomial space.

For the if part, let $A \in$ PSPACE. For every input length n we construct an aggregate D_n that has the desired properties. The computation of D_n on input $x \in \{0,1\}^n$ consists of two phases. First, x is mapped to a quantified boolean formula Φ_x that is true iff $x \in A$. In a second phase, D_n examines Φ_x and outputs 1 iff Φ_x is true.

The set of true quantified boolean formulas is PSPACE-complete with respect to logspace many-one reductions [SM73]. Therefore the mapping of inputs x to formulas Φ_x reduction can be realized by a layered polynomial-size circuit C_n. We use C_n as a component of D_n to implement the first phase of the computation. The input x is fed into C_n, which outputs Φ_x in every step from step $p(n)$ onwards. Here $p(n)$ is the depth of C_n. We may assume that Φ_x is in prenex form with an even number of alternating quantifiers, beginning with an existential quantifier. Thus $\Phi_x = \exists y_1 \forall y_2 \exists y_3 \cdots \forall y_m \; \phi(y_1, \ldots, y_m)$ for some quantifier-free ϕ. This form of Φ_x canonically yields a circuit that decides whether Φ_x is true or not; see [Pap94].

The circuit, see Figure 3, has the following form. It is a balanced binary tree with $m+1$ levels, one for each quantifier in ϕ, plus the leaf level. The leaves are the input gates and the root is the output gate. The input of the leaf at path $b_1 \cdots b_m$ is $\phi(b_1, \ldots, b_m)$ is $\phi(b_1, \ldots, b_m)$, i.e., the value of ϕ when the variables $y_1, \ldots, y_m$ are assigned $b_1 \cdots b_m$. The nodes inside the tree are or-gates on odd and and-gates on even levels, where the root level is numbered 1. Each level corresponds to one of the quantifiers and the circuit outputs 1 iff Φ_x is true.

This circuit cannot be used as a part of the aggregate D_n since it has exponential size. Fortunately, we are allowed to re-use gates. Instead of using 2^i or-gates resp. and-gates on level i in parallel, we can use the same gate 2^i times in order to evaluate the circuit. The values $\phi(b_1, \ldots, b_m)$ are fed successively in lexicographical order with respect to the assignments to the component shown in Figure 4. To achieve this, we feed the output of an m-bit enumerator together with the formula ϕ to a layered evaluation circuit that outputs the desired sequence of values.

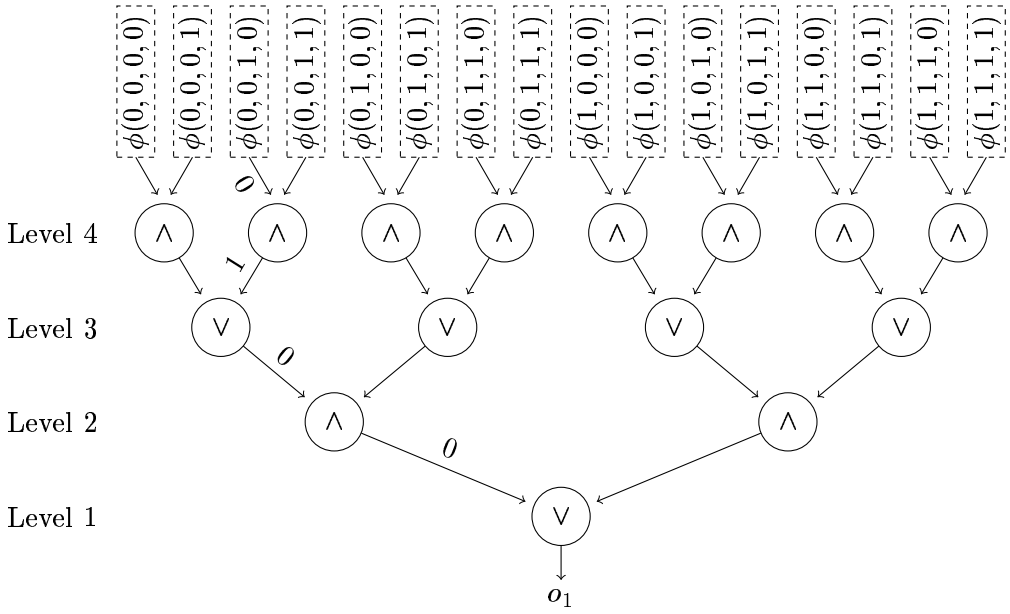

Figure 3: The circuit for verifying Φ_x with $m = 4$ quantifiers.

The first of any two values fed to the aggregate from Figure 4 has to be stored for one step, because it has to be delivered together with the next incoming value to the and-gate that processes them and that delivers its output to the next level. The output in the second step will be called a *relevant output* since it presents an output of a gate in the circuit, in this case the very left one in level m. The first of two relevant outputs has to be stored until the second is available, in order to deliver them together to the gate presenting the next level in the circuit. The same scheme is applied to each level; and thus the last or-gate in the component at last yields a relevant output, which is exactly the output of the simulated circuit.

One problem remains to be solved: It is not apparent how to build a storage device that fits the needs of our construction without using components of size at least 2. The problem is not solved directly, but circumvented. Each relevant output that has to be stored is computed twice in order to reset the storage device based on a flipflop.

Let a_1 and a_2 be a successive pair of relevant outputs, that are to be processed together. First, a_1 is stored in a flipflop. After a_2 has been delivered and processed with a_1 on the next level, a_1 is computed *once more* to reset the flipflop. Then, just to simplify the construction, a_2 is also computed again, before it is the next pair's turn.

The construction of the storage component S shown in Figure 1 guarantees that a 1 can only reach the flipflop if a signal in the second input indicates a relevant output from the previous level. Without the recomputation we could generate the needed sequence of assignments for m variables by an m-bit enumerator. Now we have to guarantee that the feeding of the values $\phi(b_1, \ldots, b_m)$ is done in the right order.

When a pair (a_1, a_2) of relevant outputs on level i has to be computed a second time, the sequence of assignments used to obtain those values has to be fed in a second time as well. That sequence contains exactly those assignments that have $b_1 \cdots b_{i-1}$ as prefix, where $b_1 \cdots b_{i-1}$ is the path to the gate in the simulated circuit that gets a_1 and a_2 as input. We can thus use an $(m+1)$-bit

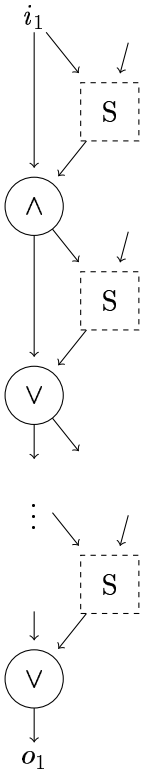

Figure 4: The aggregate replacing the circuit of Figure 3.

enumerator and skip the $(i-1)$-th bit of the output to obtain an m-bit enumerator which repeats that sequence one time. As on each level the relevant outputs have to be computed twice, we use an $2m$-bit enumerator and skip one bit out of every two bits and keep only the second.

All parts of the aggregate, namely the reduction circuit, the enumerator, the evaluation circuit, and the aggregate in Figure 4, are of size polynomial in n and logspace-uniform. Besides the loops used in the flipflops there are no cycles in the graph of the aggregate. □

The above proof also works for a nonuniform variant of the theorem:

Theorem 4.2. *A language $A \subseteq \{0,1\}^*$ is recognized by a family of polynomial size aggregates that are acyclic except for loops on xor-gates iff $A \in$ PSPACE/poly.*

Proof. As in the proof of Theorem 2.6 we can use the encoding of the aggregate as advice in order to simulate its computation on a Turing machine. Conversely, let $A \in$ PSPACE/poly via a set $B \in$ PSPACE and an advice function f. We can argue the same way as in the previous proof, except that $\big(x, f(|x|)\big)$ instead of x is reduced to Φ_x. □

4.2 Loops on Other Gates

In the Main Theorem 4.1 we consider aggregates where loops are only allowed on xor-gates. Does Theorem 4.1 still hold if we use another type of gates instead, nand-gates for instance? The answer is no for most the 16 types of gates with fan-in two. Only for xor- and equality-gates, which compute the negation of xor, the corresponding class is PSPACE. In all other cases we only get P. (Of course, if P = PSPACE, all gates have the same power.)

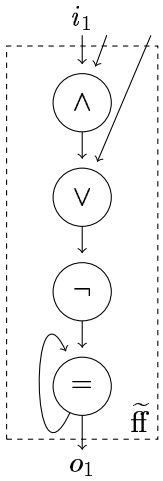

Figure 5: The component $\widetilde{\text{ff}}$. The equality-gate flips to output 1 at the beginning of the computation. Appropriate sequences are fed to the or-gate and the and-gate such that the equality-gate flips back to 0 until the simulation of the flipflop starts.

Theorem 4.3. *Let $\mathcal{F}$ be a set of functions $f\colon \{0,1\}^2 \to \{0,1\}$. Let $\mathcal{C}_\mathcal{F}$ denote the class of languages that can recognized by a logspace-uniform family of polynomial-size aggregates that are acyclic except for loops on the f-gates, $f \in \mathcal{F}$. Then*

1. *$\mathcal{C}_\mathcal{F} =$ PSPACE, if the xor-function or the equality-function belongs to $\mathcal{F}$,*
2. *$\mathcal{C}_\mathcal{F} =$ P, otherwise.*

Proof. For the first statement we only need to show PSPACE $\subseteq \mathcal{C}_\mathcal{F}$, where $\mathcal{F}$ contains the equality-function, but not the xor-function. In order to show this, the construction of the aggregate D_n from the proof of Theorem 4.1 is altered the following way.

We replace every flipflop ff by a small component $\widetilde{\text{ff}}$ that simulates ff with a delay of $r = 3$ steps starting in step $t = 4$. That is, if ff maps an input bit sequence $(e_0 e_1 \dots)$ to an output bit sequence $(a_0 a_1 \cdots)$, then $\widetilde{\text{ff}}$ maps $(***e_0 e_1 \dots)$ to $(*******a_0 a_1 \cdots)$. The stars represent arbitrary values from $\{0,1\}$. Figure 5 shows how to build $\widetilde{\text{ff}}$ using looped equality-gates.

Outside the components $\widetilde{\text{ff}}$, we have to balance the delay r by inserting additional identity-gates. Consider a node g in D_n whose output has at least two branches, one of which leads without further branching to a flipflop ff, which is to be replaced by $\widetilde{\text{ff}}$. We insert three identity-gates between g and any of the other branches. Note that this method of delay balancing makes use of the special structure of D_n.

For the second statement, the inclusion P $\subseteq \mathcal{C}_\mathcal{F}$ can easily be obtained using the well-known circuit characterization of P. We sketch how to show $\mathcal{C}_\mathcal{F} \subseteq$ P. As above, we may consider the looped gates of the aggregate D_n as special gates with fan-in 1, so that the graph of the aggregate D_n becomes acyclic. For a gate v, let depth(v) be defined as for circuits in Section 2. We will argue that, since the maximum depth of a node in D_n is bounded by D_n's size, the computation time is bounded by a polynomial. Therefore the computation can be simulated by a polynomial-time Turing machine.

We say gate g has *period* 2 from step t on, if g, from step t on, outputs repeatedly the sequence ab for some $a, b \in \{0,1\}$. If all parent gates of a gate g have period 2 from step t on, then g itself has period 2 from step $t+1$. By easy, but rather tedious calculations it can be shown that this also holds for the special gates, provided the underlying looped gate is neither an xor-gate nor an

equality-gate. Since all gates of depth 0 have period 1 from step 1, we conclude that the gates of layer k have period 2 from step k. Therefore, if the flag-gate does not output 1 within the first $k+1$ steps, it will never do so. □

5 Using Leaf Language Characterizations

In this section we show that a weaker result than Theorem 4.1 can be obtained with much less effort, using the leaf language characterization of PSPACE given in [HLS+93]. Hertrampf et al. have shown that PSPACE has a regular leaf language. We use this to give an easy proof that every language in PSPACE can be decided by an logspace-uniform aggregate of polynomial size where the component size is bounded by a fixed constant.

For the concept of leaf languages, consider a nondeterministic polynomial-time Turing machine M that is *normalized*. This means that for every input string $x \in \{0,1\}^n$ with $n \in \mathbb{N}$ there are exactly $p(n)$ binary nondeterministic branchings, where p is a polynomial. Let $M(x,y)$ be the result on input x and with $y \in \{0,1\}^{p(n)}$ as coding of the $p(n)$ nondeterministic decisions. For a given pair $(x,y) \in \{0,1\}^n \times \{0,1\}^{p(n)}$ the result $M(x,y)$ can be deterministically computed in polynomial time. Let $\mathrm{leaf}_M(x) = \big(M(x,y)\big)_{y \in \{0,1\}^{p(n)}}$ be the string of the results on input x where the y's are taken from $\{0,1\}^{p(n)}$ in lexicographical order. For any language $L \subseteq \{0,1\}^*$, the class $\mathrm{LEAF}(L)$ contains all languages A for which there exists a normalized polynomial-time Turing machine M such that $x \in A$ iff $\mathrm{leaf}_M(x) \in L$.

Fact 5.1 ([HLS+93]). *There is a regular language R such that* $\mathrm{LEAF}(R) = \mathrm{PSPACE}$.

Theorem 5.2. *There isa $k \in \mathbb{N}$ such that every language $A \in$ PSPACE can be recognized by a family of logspace-uniform polynomial-size aggregates that have component size at most k.*

Proof. Let $A \in \mathrm{PSPACE} = \mathrm{LEAF}(R)$ via a normalized polynomial-time Turing machine M_A that uses $p(n)$ binary decisions for inputs of length n. We can assume $R \subseteq \{0,1\}^*$. Let R be accepted by a deterministic finite automaton M_R.

The aggregate for input length n is constructed as follows. A $p(n)$-bit enumerator successively produces all strings $y \in \{0,1\}^{p(n)}$ in lexicographical order. Together with the input x, they are fed to a circuit that computes $M_A(x,y)$. Then this circuit will output the sequence $\mathrm{leaf}_{M_A}(x)$ as the values for y are enumerated. This sequence is fed to a component B that simulates M_R. It processes one input symbol, i. e., one bit, within 2^k steps for some $k \in \mathbb{N}$. By using a $(p(n)+k)$-bit enumerator instead of the $p(n)$-bit enumerator and skipping the k first bits of its output, we make sure that only every 2^k steps a new bit of the sequence $\mathrm{leaf}_{M_A}(x)$ is delivered to B.

When the last bit of sequence has been processed, the output of B is 1 iff the input x was in A. □

6 Conclusion and Open Problems

The essential properties of the aggregate model are that

1. every gate can be used many times and
2. intermediate results can be fed back to the components that produced them.

In the case of polynomial-size aggregates we have seen that the second property can almost entirely be abandoned without any loss computational power.

In this work we focused on the computational power of aggregates with component size one. Theorem 4.3 tells us that the computational power depends on which type of gates are allowed to have loops, unless P = PSPACE. If we consider component size $c > 1$, one may ask again: Which components of size c yield exactly PSPACE as opposed to P? If we consider a component that can be used to simulate flipflops, it is one of them. However, if B cannot simulate flipflops, can we conclude that any gate will have period 2?

The component size of a graph is not its only parameter of interest. We suggest to investigate which other graph parameters affect the computational power of aggregates. A natural candidate is, for instance, the depth of an aggregate. Can we characterize P in terms of aggregates of logarithmic depth?

References

[Ata98] Mikahail Atallah, editor. *CRC Handbook on Algorithms and Theory of Complexity.* CRC Press, 1998.

[BDG95] José Balcázar, Josep Díaz, and Joaquim Gabarró. *Structural Complexity I.* Springer-Verlag, 1995.

[DC80] Patrick W. Dymond and Stephen A. Cook. Hardware complexity and parallel computation. *Proceedings of the IEEE 21st Annual Symposium on Foundations of Computer Science*, pages 360–372, 1980.

[DC89] Patrick W. Dymond and Stephen A. Cook. Complexity theory of parallel time and hardware. *Information and Computation*, 80:205–226, 1989.

[HLS+93] Ulrich Hertrampf, C. Lautemann, T. Schwentick, H. Vollmer, and K. W. Wagner. On the power of polynomial bit-reductions. *Proceedings of the 8th Structure in Complexity Theory*, pages 200–207, 1993.

[Hon86] J. W. Hong. *Computation: Computability, Similarity and Duality.* John Wiley and Sons, 1986.

[KL80] R. Karp and R. Lipton. Some connection between non-uniform and uniform complexity classes. *Proceedings of the 12th ACM Symposium on Theory of Computing*, pages 302–309, 1980.

[Pap94] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Sav72] E. Savage. Computational work and time of finite machines. *Journal of the ACM*, 19:660–674, 1972.

[SM73] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. *Proceedings of the 5th ACM Symposium on the Theory of Computing*, pages 1–9, 1973.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'