Incremental branching programs

Anna Gál* Univ. of Texas at Austin

Michal Koucký† CWI, Amsterdam

Pierre McKenzie‡ Université de Montréal

December 16, 2005

Abstract

We propose a new model of restricted branching programs which we call *incremental branching programs*. This is in line with the program proposed by Cook in [Co74] as an approach for separating the class P of problems solvable in polynomial time from the class L of problems solvable in logarithmic space, focusing on the P-complete problem GEN and increasingly stronger models of computation. As in other restricted branching program models, we consider syntactic and semantic versions of the model. We show that *syntactic* incremental branching programs capture previously studied structured models of computation for GEN, namely marking machines [Co74] and Poon's extension [Po93] of jumping automata on graphs [CoRa80]. We then prove exponential size lower bounds for our syntactic incremental model, and for some other restricted branching program models as well. We further show that nondeterministic syntactic incremental branching programs are provably stronger than their deterministic counterpart when solving a natural NL-complete GEN subproblem. It remains open if syntactic incremental branching programs are as powerful as unrestricted branching programs for GEN problems. So far we are unable to prove lower bounds for *semantic* incremental branching programs, and we propose studying this model as a natural next step in the quest to understand the relationship between L and P.

*Dept. of Computer Science, University of Texas at Austin, Austin, TX 78712, USA. e-mail: panni@cs.utexas.edu. Supported in part by NSF Grant CCF-0430695 and an Alfred P. Sloan Research Fellowship.

†Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands. On leave from Mathematical Institute of Czech Academy of Sciences, Praha, Czech Republic. Part of this work was done while being a postdoctoral fellow at McGill University, Montréal, Québec, Canada. Supported in part by NWO vici project 2004-2009, project No. 1M0021620808 of MŠMT ČR and grant 201/05/0124 of GA ČR, Institutional Research Plan No. AV0Z10190503. e-mail: M.Koucky@cwi.nl

‡Dept. d'Informatique et recherche opérationnelle, Université de Montréal, C.P. 6128, succ. Centre-Ville, Montréal (Québec), H3C 3J7 Canada. e-mail: mckenzie@iro.umontreal.ca Supported by the NSERC of Canada and the (Québec) FQRNT

1 Introduction

Is the complexity class L of the problems solvable in deterministic logarithmic space properly contained in the class P of problems solvable in polynomial time? This question arose in the late 1960's [Co71] and remains open today. As is well known, L is captured by polynomial size branching programs. To separate L from P, it thus suffices to identify a language in P that no polynomial size branching program can recognize.

In this paper we focus on the problem GEN, a P-complete problem having natural NL-complete and L-complete subproblems [Co74, JoLa77, BaMc91]. We consider a restriction on branching programs, that all currently known size-efficient branching programs solving GEN and its subproblems seem to possess. We call branching programs obeying this restriction *incremental branching programs*. As in other restricted branching program models, we consider syntactic and semantic versions of the model: the syntactic restriction imposes a condition on all the graph-theoretic paths in the branching program, and the semantic restriction imposes a condition only on the paths actually traversed by an input. We prove exponential size lower bounds for syntactic incremental branching programs computing GEN. We also obtain exponential lower bounds for some other restricted branching program models for computing GEN, that can be viewed in a general common framework with incremental branching programs, but do not require the incrementality restriction. We refer to this more general framework as *tight computation*. This framework is specific to the n-way branching program model of [BoCo82]. The models captured by this framework include read once branching programs and an extension of monotone nondeterministic branching programs to n-way branching programs (called S*-monotone programs* where S is a subset of the possible values of the variables), in addition to incremental branching programs.

Why should one consider a new branching program restriction when so many restrictions have been investigated (see for example [We00])? Here are our main reasons:

1. The model that we propose offers a new perspective on the known structured lower bounds for GEN. Indeed we show that syntactic incremental branching programs capture the strength of Cook's marking machines [Co74] when solving GEN and they capture the strength of Poon's NNJAG extension [Po93] of Cook and Rackoff's jumping automata on graphs [CoRa80] when solving the NL-complete problem $\text{GEN}_{(2rows)}$. Hence size bounds for one model can be carried to the other and back. An ingredient in proving this equivalence is a "Symmetrization lemma" that hinges on identifying, given an incremental branching program, a set of nodes that make "balanced incremental progress" towards solving a large enough subset of all GEN instances (see Lemma 5.2).

2. Our analysis of incremental branching programs reveals that *any* purported subexponential-size branching program solving GEN would contain some easily identifiable edges that make no local progress for the computation but that must contribute crucially at a higher description level (see Remark 4.5). Hence the analysis of incremental branching programs can lead to new insights into computation of unrestricted branching programs.

3. All currently known upper bounds for GEN and its various subproblems can be achieved by syntactic incremental branching programs, and it remains open if syntactic incremental branching programs are as powerful as unrestricted branching programs for GEN problems.

4. While so far we have not been able to analyze them, semantic incremental branching programs may provide the answer to Cook's [Co74] and Edmonds' [EPA99] requests for a computational model intermediate between marking machines and NNJAG's on the one hand, and unrestricted branching programs on the other.

We view the above as good reasons to study incremental branching programs. In this paper we show that strong exponential size lower bounds for syntactic incremental branching programs for GEN follow from [Co74, PTC77] via our Symmetrization lemma, and slightly weaker lower bounds can be derived from monotone circuit depth lower bounds [RaMc99], revealing an informative connection. In particular, marking machine lower bounds (albeit weaker than those from [Co74, PTC77]) follow from monotone circuit depth lower bounds. Figure 1 summarizes our main bounds on the sizes of syntactic incremental branching programs and $\{1\}$-monotone nondeterministic n-way branching programs solving the $n \times n$ instances of the P-complete problem GEN, of the NL-complete GEN restriction $\text{GEN}_{(2rows)}$ and of the L-complete problem $\text{GEN}_{(1row)}$. See sections 2 and 3 for the definition of GEN and the precise definitions of our branching program models.

The most size-efficient branching programs that we know how to obtain for GEN and its various subproblems are indeed syntactic incremental. As Figure 1 shows, their size is outright exponential for GEN. For $\text{GEN}_{(2rows)}$, we note a super-polynomial separation between the power of deterministic and nondeterministic syntactic incremental branching programs.

	Deterministic syntactic incremental	Nondeterministic syntactic incremental	Nondeterministic {1}-monotone
GEN	$O(n^2 2^n) \cap \Omega(2^{cn/\log n})$	$O(2^n) \cap \Omega(2^{cn/\log n})$	$O(2^n) \cap \Omega(2^{n^\delta})$
$\text{GEN}_{(2rows)}$	$n^{\Theta(\log n)}$	$O(n^2)$	$O(n^2)$
$\text{GEN}_{(1row)}$	$O(n^2)$		

Figure 1: Main size bounds presented in this paper. Here δ and c are specific constants.

From the technical perspective our paper does not develop major new lower bound techniques. Our proofs are based on reductions of varying degrees of difficulty and they appeal to the lower bounds from [Co74, PTC77, RaMc99, EPA99]. We now face a significant new open question: how powerful are semantic incremental branching programs? Are $f(n)$-size semantic incremental branching programs strictly more powerful than $O(f(n))$-size syntactic incremental branching programs? Can unrestricted (n-way) branching programs for GEN be simulated by semantic, or even by syntactic incremental branching programs without a significant size blowup? It should be noted that in the context of read-k-times branching programs, the semantic variant is provably much more powerful than its syntactic counterpart [Ju95, BJS01], indicating that semantic incremental branching programs may also behave quite differently from their syntactic counterparts. We believe that these questions should be investigated further.

2 Preliminaries

2.1 GEN problems

We write $[n]$ for $\{1, 2, \ldots, n\}$. When $T \subseteq [n] \times [n] \times [n]$ and $S \subseteq [n]$, we write $\langle S \rangle_T$ for the *closure of S under T*, defined as the smallest $S' \supseteq S$ such that the following holds for every $(i, j, k) \in [n] \times [n] \times [n]$: if $i \in S'$ and $j \in S'$ and $(i, j, k) \in T$ then $k \in S'$. We will work with the following problems:

Problem GEN
Given: A function $g : [n] \times [n] \to [n]$ prescribing $T^g \subseteq [n] \times [n] \times [n]$.
Determine: Whether $n \in \langle \{1\} \rangle_{T^g}$.

Problem RELGEN
Given: An n^3-length bit string prescribing a set $T \subseteq [n] \times [n] \times [n]$.
Determine: Whether $n \in \langle \{1\} \rangle_T$.

We will talk about n-GEN and n-RELGEN when only a particular value n is considered, this will be necessary since we work in nonuniform models. (The name GEN comes from "Generating Problem" and the name RELGEN comes from the *relational* version of the Generating Problem.) We will view n-RELGEN as a Boolean function of n^3 variables (n-RELGEN : $\{0,1\}^{n^3} \to \{0,1\}$), and n-GEN as a function over n^2 n-ary variables (n-GEN : $[n]^{n^2} \to \{0,1\}$). Note that n-RELGEN is a monotone Boolean function while the Boolean version of n-GEN (over $n^2 \log_2 n$ variables obtained by encoding the values of the n^2 n-ary variables as $\log_2 n$-length bit strings) is not monotone. We call an instance T of GEN or RELGEN *positive* if $n \in \langle \{1\} \rangle_T$ otherwise the instance T is *negative*. It is known that

- GEN (and thus RELGEN) is P-complete [Co74, JoLa77],
- $\text{GEN}_{(2rows)}$, namely the restriction of GEN in which $i * j \neq 1 \Rightarrow i \leq 2$, is NL-complete [BaMc91],
- $\text{GEN}_{(1row)}$ namely the restriction of GEN in which $i * j \neq 1 \Rightarrow i = 1$, is L-complete [BaMc91].

Fix $n > 0$. We call $[n]$ the set of n-GEN elements. Both an n-GEN instance and an n-RELGEN instance define a set of triples $(i, j, k) \in [n] \times [n] \times [n]$ which we denote $i * j = k$ even when k is not uniquely defined from i and j. Recall that an n-GEN instance is defined by a function $g : [n] \times [n] \to [n]$, thus the corresponding set of triples T^g has the property that for each pair $(i, j) \in [n] \times [n]$ there is exactly one value $k \in [n]$ such that $(i, j, k) \in T^g$. On the other hand, n-RELGEN instances may involve arbitrary sets $T \subseteq [n] \times [n] \times [n]$, including the possibility that some $i * j$ is not assigned any value, that is $(i, j, k) \notin T$ for any k. (The name RELGEN indicates that the underlying set of triples corresponds to a relation, rather than a function.)

2.2 Branching programs

Since n-GEN is defined over n-ary input variables, it is convenient for us to work with n-way branching programs, first defined by Borodin and Cook [BoCo82]. We also need the nondeterministic extension of the model, defined by Borodin, Razborov and Smolensky [BRS93].

Definition 2.1 *[BRS93] A* nondeterministic n-way branching program *is a directed acyclic rooted multi-graph with a distinguished sink node labeled ACCEPT. The edges out of non-sink nodes are either unlabeled, or labeled "$x_i = j$" for some variable x_i and $j \in [n]$. Only inputs satisfying the statement on the label may follow the labeled edges, all inputs are allowed to follow the unlabeled edges. An input $x_1, \ldots, x_t$ is* accepted *by the program if there is at least one directed path leading from the root to the ACCEPT node, such that the input $x_1, \ldots, x_t$ is allowed to follow it. (As there may be multiple edges between two nodes in a branching program by a path we always understand a sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{m-1}, v_m)$ rather than just a sequence of vertices $v_1, v_2, \ldots, v_m$.) A nondeterministic n-way branching program* computes *a function $f : [n]^t \to \{0, 1\}$ if $f(x_1, \ldots, x_t) = 1$ if and only if $x_1, \ldots, x_t$ is* accepted *by the program. A* deterministic n-way branching program *must satisfy the additional restrictions that it has no unlabeled edges, and there are exactly n edges out of each non-sink node with the n possible labels $x = j$ for $j = 1, \ldots, n$ for the same variable x.*

The *size* of a branching program is the number of its nodes. If the program contains other sink nodes in addition to the ACCEPT node, they are labeled REJECT. Note that deterministic programs computing non-constant functions have at least one REJECT node, but REJECT nodes may be omitted from nondeterministic programs. Note also that the nondeterministic model defined above is usually called a switching-and-rectifier network if the underlying graph is not required to be acyclic.

A path is called *consistent* if for every variable x_i and for all values $j_1 \neq j_2$, the labels $x_i = j_1$ and $x_i = j_2$ do not both appear on the path. Note that since no input will follow an inconsistent path, the correctness of the program in itself gives no requirements for inconsistent paths.

Definition 2.1 contains as a special case (deterministic or nondeterministic) Boolean branching programs if $n = 2$ (using the values 0 and 1 instead of 1 and 2, of course). Several slightly different definitions of nondeterministic branching programs have appeared in the literature, in particular they may involve introducing guessing nodes. The size necessary to compute a given function under these different definitions remains polynomially related (see e.g. [Ra91]), note however that this is not automatically inherited in various restricted versions of the models.

For an n-ary function $f : [n]^t \to \{0, 1\}$ let $f_{bin} : \{0, 1\}^{t \log_2 n} \to \{0, 1\}$ be the Boolean function obtained from f by encoding its variables as binary strings. If f can be computed by n-way branching programs of size $s(n)$ then f_{bin} can be computed by Boolean branching programs of size $n \cdot s(n)$. No size increase occurs in the reverse direction. Thus, proving super-polynomial size lower bounds for deterministic or nondeterministic n-way branching programs computing n-GEN would separate P from L or NL, respectively.

3 Definitions

For $i \in [n]$, we write $\chi_n(i)$ for the n-bit string $0^{i-1}10^{n-i}$ (i.e. the characteristic vector of the singleton set $\{i\}$), and we write $\chi(i)$ for $\chi_n(i)$ when n is understood. Given an n-GEN instance g, we write $\chi_n(g)$ (or $\chi(g)$ when n is understood) for the length-n^3 n-RELGEN instance $\chi(g(1, 1))\chi(g(1, 2)) \cdots \chi(g(n, n))$. Note that for any n-GEN instance g, $\text{GEN}(g) = \text{RELGEN}(\chi_n(g))$.

An n-RELGEN instance $w \in \{0, 1\}^{n^3}$ is considered as divided up into n^2 n-bit blocks denoted w_{i*j} for $(i, j) \in [n] \times [n]$ and concatenated to form w. We will refer to the Boolean variables of n-RELGEN as $w_{i,j,k}$.

A triple of the form $i*j = 1$ for some i and j is called a *trivial* triple (since the element 1 is always trivially included in the closure $\langle\{1\}\rangle_T$). For an n-RELGEN instance $w \in \{0, 1\}^{n^3}$, we write $\text{TRIVEXT}(w)$ (for *trivial extension*) to mean $w \vee (10^{n-1})^{n^2}$, that is, the n-RELGEN instance obtained from w by adding to the set of triples represented by w all the trivial triples, setting $w_{i,j,1} = 1$ for each $i, j \in [n]$.

A block w_{i*j} is said to be *heavy* if two (or more) of its bits are 1.

3.1 Tight computation of GEN

Given a (deterministic or nondeterministic) n-way branching program P computing n-GEN, we denote by $\text{MON}(P)$ the nondeterministic *Boolean* branching program obtained from P as follows: replace each edge label of the form $i * j = k$ of P by the label $w_{i,j,k} = 1$.

Since $\text{MON}(P)$ uses only edge labels of the form $x = 1$, it computes a monotone Boolean function, which we denote by $f_{\text{MON}(P)}$. Note that for any n-GEN instance g, we have

$$\text{GEN}(g) = \text{RELGEN}(\chi(g)) = f_{\text{MON}(P)}(\chi(g)).$$

On the other hand, the fact that P computes n-GEN in itself does not place any requirements on the value of $f_{\text{MON}(P)}$ over inputs $w \in \{0,1\}^{n^3}$ that are not of the form $\chi(g)$ for any n-GEN instance g. In particular, while we know that $\text{MON}(P)$ computes a monotone Boolean function that agrees with RELGEN on inputs of the form $\chi(g)$, we have no reason to expect that $\text{MON}(P)$ would actually compute n-RELGEN, or even agree with n-RELGEN on any other inputs except what is implied by the monotonicity of $f_{\text{MON}(P)}$.

It turns out that the following additional requirement on n-way branching programs computing n-GEN is sufficient to obtain exponential lower bounds. We require that in addition to inputs of the form $\chi(g)$, $f_{\text{MON}(P)}$ agrees with RELGEN also on the trivial extensions $\text{TRIVEXT}(\chi(g))$. Since $f_{\text{MON}(P)}$ is monotone, this is equivalent to just requiring that if $f_{\text{MON}(P)}(\chi(g)) = 0$ then $f_{\text{MON}(P)}(\text{TRIVEXT}(\chi(g))) = 0$ as well. Thus, we just require that for any n-GEN instance g that has no accepting path in P, no path of P can reach the ACCEPT node if the only edges used in addition to the edges that g could traverse are labeled by trivial triples of the form $i * j = 1$.

Definition 3.1 *We say that a (deterministic or nondeterministic) n-way branching program P* tightly computes *n-GEN if it computes n-GEN, and for any n-GEN instance g*

$$\textit{if } f_{\text{MON}(P)}(\chi(g)) = 0 \textit{ then } f_{\text{MON}(P)}(\text{TRIVEXT}(\chi(g))) = 0.$$

Definition 3.2 *We say that a Boolean function $f : \{0,1\}^{n^3} \to \{0,1\}$* represents *$n$-GEN if $f(\chi(g)) = RELGEN(\chi(g))$ for any n-GEN instance g.*

We say that a Boolean function $f : \{0,1\}^{n^3} \to \{0,1\}$ tightly represents *n-GEN if it represents n-GEN and $f(\text{TRIVEXT}(\chi(g))) = RELGEN(\text{TRIVEXT}(\chi(g)))$ for any n-GEN instance g.*

The following is immediate from the above definitions:

Proposition 3.3 *An n-way branching program P computes n-GEN if and only if $f_{\text{MON}(P)}$ represents n-GEN, and an n-way branching program P tightly computes n-GEN if and only if $f_{\text{MON}(P)}$ tightly represents n-GEN.*

Note that in models where inconsistent paths are excluded, for example in deterministic read-once branching programs, tight computation is automatically guaranteed. Next we define two versions of the model that guarantee tight computation of GEN - without excluding inconsistent paths in general: $\{1\}$-monotone nondeterministic n-way branching programs and syntactic incremental branching programs.

3.2 Monotone nondeterministic n-way branching programs

In the case of nondeterministic Boolean branching programs, Definition 2.1 can easily be modified to define *monotone* nondeterministic Boolean branching programs, by simply forbidding labels of the form $x = 0$ (using the values 0 and 1 instead of 1 and 2, of course), see e.g. [Ra91].

It is not clear what would be the analogous restriction for nondeterministic n-way branching programs when $n > 2$. Here we consider a restriction on nondeterministic n-way branching programs that extends the definition of monotone nondeterministic Boolean branching programs to $n > 2$. Similarly to the Boolean case, we simply forbid some of the n possible values to be used in edge labels. Just like in the case of Boolean branching programs, "monotonicity" makes more sense in the nondeterministic framework, and not every n-ary function can be computed under this restriction.

Definition 3.4 *Let $\emptyset \neq S \subset [n]$. A nondeterministic n-way branching program is S-monotone if edge labels $x = j$ with $j \in S$ do not appear in the program.*

Definition 3.5 *Let $x, y \in [n]^t$ and $\emptyset \neq S \subset [n]$. We say that $x <_S y$ if for any $i \in [t]$ either $x_i = y_i$ or $x_i \in S$ and $y_i \notin S$. A function $f : [n]^t \to \{0,1\}$ is S-monotone, if for any $x, y \in [n]^t$ such that $x <_S y$, we have $f(x) \leq f(y)$.*

Note that our definition of S-monotone functions includes both monotone and anti-monotone Boolean functions as a special case. Clearly, S-monotone branching programs can compute only S-monotone functions. Notice that GEN is a $\{1\}$-monotone function, and the variant of GEN where the problem is to determine whether n is in the closure of some fixed starting set S is an S-monotone function. We can match the current best upper bounds for various GEN subproblems by $\{1\}$-monotone nondeterministic n-way branching programs without significant increase in size. That is, so far we have no results separating the power of monotone and non-monotone nondeterministic n-way branching programs for computing GEN.

3.3 Incremental branching programs

It seems natural for an n-way branching program solving n-GEN to try to find the elements in the closure $\langle\{1\}\rangle_{T^g}$ "incrementally", that is by asking questions $i * j = ?$ only for elements i, j already known to be in $\langle\{1\}\rangle_{T^g}$. In fact all the constructions (known to us) that achieve the current best upper bounds for various subproblems of GEN have this property. We formally define the incrementality property below.

Let P be a (deterministic or nondeterministic) branching program computing n-GEN. For each vertex u of P, we will define the set $A(u)$ ("available set") of elements that have already been generated along *every* path reaching u.

Given a path π from the root to some vertex u in P, let T^π be the set of triples that appear as edge labels along π. Let $\text{PATHS}(u)$ denote the set of all graph theoretic paths in P starting from the root and reaching u. Let

$$A(u) = \bigcap_{\pi \in \text{PATHS}(u)} \langle\{1\}\rangle_{T^\pi}.$$

We obtain a potentially larger set, if we only require that its elements are generated along every path reaching u that may be followed by some GEN instance. Recall that a path in P is consistent if for every pair (i, j) and for all values $k_1 \neq k_2$, the labels $i * j = k_1$ and $i * j = k_2$ do not both appear along the path. If π is consistent then $T^\pi \subseteq T^g$ for some n-GEN instance g. On the other hand, no GEN instance can follow an inconsistent path. We denote by $\text{GENPATHS}(u)$ the set of all consistent paths starting from the root and reaching u. Let

$$A_{\text{GEN}}(u) = \bigcap_{\pi \in \text{GENPATHS}(u)} \langle\{1\}\rangle_{T^\pi}.$$

Note that $\text{GENPATHS}(u) \subseteq \text{PATHS}(u)$, and thus $A(u) \subseteq A_{\text{GEN}}(u)$.

Definition 3.6 *A (deterministic or nondeterministic) n-way branching program for n-GEN is* (semantic) incremental *if for every edge with label $i * j = k$ directed out of a node u the condition $\{i, j\} \subseteq A_{GEN}(u)$ holds. The program is* syntactic incremental *if for every edge with label $i * j = k$ directed out of a node u the stronger condition $\{i, j\} \subseteq A(u)$ holds.*

Let P be a branching program that computes n-GEN and let $\pi = (v_0, v_1), (v_1, v_2), \ldots, (v_\ell, v_{\ell+1})$ be a path in P. In particular, we allow $\ell = 0$ so the path may consist of a single edge. We say that $i \in [n]$ is *useful* for π if the last edge $(v_\ell, v_{\ell+1})$ of π is labeled by $i * j = k$ or by $j * i = k$ for some $j, k \in [n]$, and none of the edges (v_t, v_{t+1}) for $t \in \{0, \ldots, \ell - 1\}$ is labeled by $k * j = i$ for any $j, k \in [n]$. For a node u of P, let $U(u)$ be the set of elements that are useful for some path π starting in u and leading to an arbitrary node of P. Notice, the program P is syntactic incremental if and only if $U(u) \subseteq A(u)$ for every node u of P. We denote by $\max U(P)$ the maximum size of $U(u)$ for any node u in P.

4 Lower bounds derived from monotone circuit depth bounds

We will need the following well known statement. For completeness, we include a proof in the Appendix.

Proposition 4.1 *Let $f : \{0,1\}^t \to \{0,1\}$ be a monotone Boolean function computed by a monotone nondeterministic Boolean branching program of size $s(t)$. Then f can be computed by a monotone Boolean circuit with fan-in 2 AND/OR gates in depth $O((\log_2 s(t))^2)$.*

4.1 Lower bounds in models without requiring incrementality

By the definitions of the previous section, every n-way branching program P computing GEN has an associated monotone nondeterministic Boolean branching program $\text{MON}(P)$ computing some function $f_{\text{MON}(P)}$ that represents GEN. Thus, by Proposition 4.1, proving that every monotone function representing GEN requires large monotone circuit depth would be sufficient to obtain lower bounds for unrestricted (deterministic or nondeterministic) n-way branching programs computing GEN. We can even define specific monotone Boolean functions representing GEN such that monotone circuit depth lower bounds for them would imply lower bounds for every monotone function representing GEN[1]. Unfortunately, we do not know how to prove monotone circuit depth lower bounds for these functions.

[1]For example, the n^2-th slice function of n-RELGEN has this property. See e.g. [Be81, We87] for more on slice functions.

However, the monotone circuit depth lower bounds of [RaMc99] for the RELGEN function are sufficient to derive the following statement.

Theorem 4.2 *For some $\gamma > 0$ and any t large enough, any function $f : \{0,1\}^t \to \{0,1\}$ that tightly represents GEN requires monotone circuits of depth t^γ.*

Proof. Let $\delta > 0$ be such that n-RELGEN requires monotone circuit depth n^δ (as proved in [RaMc99][Corollary 3.6]). Consider an n-RELGEN instance $w \in \{0,1\}^{n^3}$. We claim that a monotone projection can produce an m-RELGEN instance $w' \in \{0,1\}^{m^3}$, for $m = n^2 + 2n$, such that RELGEN$(w) = f(\textsc{trivext}(w'))$.

We conclude the proof of the theorem from the claim as follows. Suppose to the contrary that f can be computed in monotone depth t^γ for $\gamma = \delta/9$. On input $w \in \{0,1\}^{n^3}$, we apply the monotone projection above combined with another monotone projection computing $\textsc{trivext}(w')$ and we feed the result into the circuit for f (over inputs of length $t = m^3$). This construction gives a monotone circuit with n^3 Boolean inputs of depth $(m^3)^\gamma < (n^{2.1})^{3\gamma} < n^\delta$ computing n-RELGEN, contradicting the fact that n-RELGEN requires monotone depth n^δ.

It remains to prove the claim. To construct w' we introduce dummy elements $n+1, n+2, \ldots, m = n^2 + 2n$ that will simulate the effect of a heavy block w_{i*j} without the need for heavy blocks in w'. The elements $n+1, n+2, \ldots, 2n$ will be generated from the RELGEN source 1 in all cases:

$$w'_{1*1} = \chi_m(n+1), w'_{1*(n+1)} = \chi_m(n+2), \ldots, w'_{1*(2n-1)} = \chi_m(2n).$$

Then, for $1 \le i, j \le n$, writing $t_{ij} = 2n+(j-1)n+i$ for convenience, we coerce each former w_{i*j} block into generating the single element t_{ij} (except for t_{11}) for $(1,1) \neq (i,j) \in [n] \times [n]$:

$$w'_{i*j} = \chi_m(t_{ij}), w'_{(n+1)*(n+1)} = \chi_m(t_{11}).$$

Finally, we make each t_{ij} responsible, together with the elements $n+1, n+2, \ldots, n+n$ that are always available, for generating the elements implied by the former w_{i*j} as follows: for each $1 \le i, j, k \le n$ we set

$$w'_{t_{ij}*(n+k)} = (w_{i*j} \wedge \chi_n(k))0^{m-n}.$$

All the other w'_{p*q} blocks, $1 \le p, q \le m$, are set to 0^m. It should be clear that a monotone projection $\{0,1\}^n \to \{0,1\}^m$ can produce w'. Relabeling the element n as the target (instead of m) for m-RELGEN, we get RELGEN(w) = RELGEN(w').

To see that RELGEN$(w') = f(\textsc{trivext}(w'))$, define w'' from w' by adding selectively to each all-zero block w'_{p*q} the trivial triple $p * q = 1$. Clearly, RELGEN(w'') = RELGEN(w'). Now by construction, w' contained no heavy block, so that $w'' = \chi_m(g)$ for some m-GEN instance g, thus $f(w'') = RELGEN(w'')$. Since f tightly represents GEN, $RELGEN(w'') = f(\textsc{trivext}(w''))$. But $\textsc{trivext}(w'') = \textsc{trivext}(w')$, proving the claim. ∎

Remark 4.3 In the proof of theorem 4.2, w'' could not have been constructed monotonely from w'. This is what keeps us from getting the statement for any function representing GEN, which would give unrestricted branching program lower bounds for GEN.

By Propositions 3.3 and 4.1, Theorem 4.2 yields the following.

Theorem 4.4 *For some $\epsilon > 0$ and all n large enough, any (deterministic or nondeterministic) n-way branching program that tightly computes n-GEN has size 2^{n^ϵ}.*

Remark 4.5 Tight computation places no restrictions on the model itself, but instead requires correctness of the computation in a slightly stronger sense: it places requirements regarding acceptance on some graph-theoretic paths that no GEN instance would follow. Correctness in the usual sense places no requirements on the computation along such paths. However, tight computation places no requirements on paths with inconsistencies that don't involve trivial triples. Hence the significance of Theorem 4.4 is that any purported subexponential size branching program P solving GEN would need to make critical use of trivial triples, although such triples appear oblivious to any progress. More precisely, P would need to "maximize its acceptance ability" by incorporating graph-theoretic paths, inconsistent solely by virtue of their trivial triples, and carrying the trivial extension of some negative GEN instances to ACCEPT. The twisted sort of progress afforded hence by the seemingly inconsequential implications $i * j = 1$ and our resulting inability to handle these constitute the one – and major – obstacle preventing us from obtaining unrestricted lower bounds for GEN.

As noted at the end of Section 3.1, tight computation is automatically guaranteed in read-once deterministic branching programs, thus we automatically get the following corollary.

Corollary 4.6 *For some $\epsilon > 0$ and all n large enough, any read-once deterministic n-way branching program computing n-GEN has size $2^{n^{\epsilon}}$.*

We also obtain the following.

Proposition 4.7 *Any $\{1\}$-monotone nondeterministic n-way branching program computing n-GEN computes n-GEN tightly.*

Proof. We just need to prove that for any n-GEN instance g such that $\text{GEN}(g) = 0$, no path can reach the ACCEPT node if the only edges used in addition to the edges that g could traverse are labeled by trivial triples of the form $i * j = 1$. But a $\{1\}$-monotone nondeterministic n-way branching program has no edges labeled by trivial triples at all, so this cannot happen. ∎

By Theorem 4.4 this yields the following.

Theorem 4.8 *For some $\epsilon > 0$ and all n large enough, any $\{1\}$-monotone nondeterministic n-way branching program computing n-GEN has size $2^{n^{\epsilon}}$.*

4.2 Lower bounds for syntactic incremental branching programs

Proposition 4.9 *Any (deterministic or nondeterministic) syntactic incremental n-way branching program computing n-GEN computes n-GEN tightly.*

The statement will follow from the following much stronger statement.

Proposition 4.10 *Let P be a (deterministic or nondeterministic) syntactic incremental n-way branching program computing n-GEN. Then, for every $w \in \{0,1\}^{n^3}$ such that $RELGEN(w) = 0$, $f_{\text{MON}(P)}(w) = 0$ as well.*

We use the following claim, proved in the Appendix.

Claim 4.11 *Any incremental branching program (syntactic or semantic), can be transformed preserving both incrementality and correctness and without increasing the size of the program so that the following holds: all edges having the $ACCEPT$ node as their endpoint are labeled $i * j = n$ for some pair (i, j).*

Proof.(of Proposition 4.10) Let $w \in \{0,1\}^{n^3}$ such that $\text{RELGEN}(w) = 0$. First assume that w has no heavy blocks. This is possible in two cases: either $w = \chi(g)$ for some n-GEN instance g or w can be extended to $\chi(g)$ of some n-GEN instance g by adding trivial triples to the empty blocks of w. In the first case, $f_{\text{MON}(P)}(w) = 0$ must hold since $f_{\text{MON}(P)}(\chi(g)) = \text{RELGEN}(\chi(g))$ holds for every g. In the second case, $\text{RELGEN}(w) = 0$ implies $\text{RELGEN}(\chi(g)) = 0$ since $\chi(g)$ is obtained from w by adding only trivial triples and thus $f_{\text{MON}(P)}(\chi(g)) = 0$, since $\text{RELGEN}(\chi(g)) = f_{\text{MON}(P)}(\chi(g))$ for every g. Since $f_{\text{MON}(P)}$ is monotone, $f_{\text{MON}(P)}(\chi(g)) = 0$ implies that $f_{\text{MON}(P)}(w) = 0$ as well. Note that the above argument actually holds for any n-way program, we did not use incrementality yet. The harder case is when w has heavy blocks. In this case, the paths of $\text{MON}(P)$ that w can follow may correspond to inconsistent paths in P. By the claim, we can assume that all the edges reaching the ACCEPT node of P have labels of the form $i * j = n$. Thus, any graph theoretic path of P that reaches ACCEPT includes an edge with label $i * j = n$. In a syntactic incremental branching program, this means that $n \in A(ACCEPT)$, which implies that no path followed by w can reach the accept node, thus $f_{\text{MON}(P)}(w) = 0$ must hold. ∎

By Theorem 4.4 this yields the following.

Corollary 4.12 *Any (deterministic or nondeterministic) syntactic incremental n-way branching program computing n-GEN has size $2^{n^{\epsilon}}$ for some $\epsilon > 0$.*

5 Syntactic incremental branching programs and pebbling

In this section we study the relationship of syntactic incremental branching programs to previously studied computational models of marking machines and jumping automata on graphs. Marking machines were defined by Cook [Co74] as a model for computing GEN. Jumping automata on graphs were defined by Cook and Rackoff [CoRa80] as a computational model for solving graph s-t-connectivity; Poon [Po93] defined an extension of this model called node-named jumping automata on graphs (NNJAG). We show in this section that syntactic incremental branching programs can be efficiently simulated by marking machines and NNJAG's, and vice versa.

An instrumental tool in these efficient simulations is the relationship that exists between $\max U(P)$ and $size(P)$ of a branching program P. We establish that relationship in the Symmetrization Lemma. Because of the efficient simulations, known lower bounds for marking machines and node-named jumping automata on graphs provide us with lower bounds on the size of various types of syntactic incremental branching programs.

5.1 Marking machines

Marking machines were defined by Cook in [Co74]. We adapt his definition to our terminology while using a slightly different set of rules for moves of a marking machine than Cook uses in his original definition. (Note that the results in [Co74] hold with respect to this set of rules as well.) The rules we state are analogous to the pebbling rules of the games on graphs introduced by Paterson and Hewitt [PaHe70]. We discuss the difference between these rules more in the Appendix. A *marking machine* M operates on an instance $T \subseteq [n] \times [n] \times [n]$ of n-GEN. Each configuration of M is one of the subsets of $[n]$; it identifies the set of marked elements of $[n]$. The *initial configuration* of M is the empty set. At each step of a computation, M (nondeterministically) changes its configuration C to C' in one of the following ways: M marks the element 1, i.e., $C' = C \cup \{1\}$, or M removes a mark from an arbitrary element $r \in C$, i.e., $C' = C \setminus \{r\}$, or it marks an element $z \notin C$ provided that $(x, y, z) \in T$ for some $x, y \in C$, i.e., $C' = C \cup \{z\}$. A configuration C is *accepting* if $n \in C$. M *accepts input* T iff there is a sequence of configurations $C_0, C_1, \ldots, C_m$, where C_0 is the initial configuration, C_i follows from C_{i-1} by a legal move and C_m is an accepting configuration. We call the sequence $C_0, C_1, \ldots, C_m$ an *accepting computation.* We say that M *accepts* T *using only* ℓ *markers* if there is an accepting computation of M on T in which all configurations are of size at most ℓ.

We first establish the relationship between incremental branching programs and marking machines. As the main measure of size of marking machines is the number of marks used it should not come as a surprise that this measure relates to $\max U(\cdot)$ of branching programs.

Proposition 5.1

1. *If P is a (deterministic or nondeterministic) syntactic incremental n-way branching program computing n-GEN then there is a marking machine M that accepts every positive instance of n-GEN using at most* $\max U(P)$ *markers.*
2. *If M is a marking machine M that accepts every positive instance of n-GEN using at most ℓ markers then there is a nondeterministic syntactic incremental n-way branching program P of size $1 + \left(\sum_{i=1}^{\ell} \binom{n}{i}\right)^2$ that computes n-GEN.*

We present the proof in the Appendix.

This proposition means that lower bounds on the number of markers needed by a marking machine to solve arbitrary instances of n-GEN imply lower bounds on $\max U(P)$ for syntactic incremental branching programs computing n-GEN. Such lower bounds can be further translated into lower bounds on the size of these branching programs as the following lemma indicates.

Lemma 5.2 (Symmetrization lemma) *Let $k, n \geq 2$ be integers. Let P be a nondeterministic syntactic incremental kn-way branching program that computes kn-GEN. Then there is a nondeterministic syntactic incremental n-way branching program P' that computes n-GEN and such that $size(P') \leq size(P)$ and $\max U(P') \leq 2 + \log_k size(P)$.*

Proof. For an n-GEN instance $T \subseteq [n] \times [n] \times [n]$ define a kn-GEN instance $T^k \subseteq [kn] \times [kn] \times [kn]$ as follows. If $(x, y, z) \in T$, where $z < n$ then $(x, y, z) \in T^k$, if $(x, y, n) \in T$ then $(x, y, kn) \in T^k$ and finally for all $x, y, z \in [kn]$ if $x > n$ or $y > n$ then $(x, y, 1) \in T^k$. For a permutation $\pi : [kn] \to [kn]$ such that $\pi(1) = 1$ and $\pi(kn) = kn$, let $\pi(T^k)$ be the kn-GEN instance such that for any $x, y, z \in [kn]$, $(x, y, z) \in T^k$ iff $(\pi(x), \pi(y), \pi(z)) \in \pi(T^k)$. Notice, if u is a vertex of P such that $U(u) \cap \pi(\{n, \ldots, kn - 1\}) \neq \emptyset$ then u cannot be reached on any instance $\pi(T^k)$ that is obtained from some n-GEN instance T. This is because $U(u) \subseteq A(u)$, but no element of $\pi(\{n, \ldots, kn - 1\})$ can be generated along any path that can be followed by some instance of the form $\pi(T^k)$.

Consider a vertex u in P and a permutation $\pi : [kn] \to [kn]$ such that $\pi(1) = 1$ and $\pi(kn) = kn$, taken uniformly at random from among all such permutation. It is straightforward to verify that $Pr_\pi[U(u) \subseteq \pi(\{1, \ldots, n-1, kn\})] \leq \prod_{i=1}^{|U(u)|-2}(n-1-i)/(kn-1-i) \leq k^{-|U(u)|+2}$, where for the first inequality we assume that $U(u)$ can contain both elements 1 and kn. Let $\ell = 2 + \log_k size(P)$. $Pr_\pi[$ there is u in P such that $|U(u)| > \ell$ and $U(u) \cap \pi(\{n, n+1, \ldots, kn-1\}) = \emptyset] \leq size(P) \cdot k^{-(\ell+1)+2} \leq 1/k$. Hence, there exists a permutation $\pi : [kn] \to [kn]$ such that $\pi(1) = 1$ and $\pi(kn) = kn$ and for all vertices u in P, if $|U(u)| > \ell$ then $U(u) \cap \pi(\{n, n+1, \ldots, kn-1\}) \neq \emptyset$. Fix such a permutation π. Clearly, for any instance T of n-GEN, P on $\pi(T^k)$ cannot reach any vertex u with $|U(u)| > \ell$. Hence, we construct P' from P by relabeling all edges labeled by $i \star j = q$ to $\pi^{-1}(i) \star \pi^{-1}(j) = \pi^{-1}(q)$, then removing all edges that refer to elements in $\{n, \ldots, kn-1\}$ and finally relabeling all occurrences of kn to n. We also remove all nodes in P' that cannot be reached on some instance of n-GEN. Clearly, P' is still syntactic incremental and it computes n-GEN. Furthermore, for any node u in P', the size of $U(u)$ in P' is at most the size of $U(u)$ in P since removing edges cannot increase the number of useful elements. Hence, all vertices u in P' have $|U(u)| \leq \ell$. ∎

A similar symmetrization lemma holds for deterministic syntactic incremental branching programs.

In [Co74] Cook proves that there are instances of n-GEN that cannot be accepted by marking machines with $o(\sqrt{n})$ markers. This was later improved by Paul et al. in [PTC77]:

Proposition 5.3 ([PTC77]) *There is a constant $c > 0$ such that for all $n \geq 2$ there is an instance T of n-GEN that cannot be accepted by any marking machine using less than $cn/\log n$ markers.*

Note that [PTC77] gives a bound on the number of pebbles in a pebbling game on graphs, but this easily translates into marking machine lower bounds for GEN.

The previous three claims give us the following corollary.

Theorem 5.4 *There is a constant $c > 0$ such that for all n large enough if a nondeterministic syntactic incremental n-way branching program P solves n-GEN then it has size at least $2^{cn/\log n}$.*

Proof. Assume that for every constant $c > 0$ and infinitely many n, there is a nondeterministic incremental n-way branching program of size less than $2^{cn/\log n}$ that computes n-GEN. Hence, for infinitely many $m = \lfloor n/2 \rfloor$, there is a branching program P of size less than $2^{c(2m+1)/\log 2m}$ that solves $2m$-GEN. By the symmetrization lemma there is a nondeterministic syntactic incremental m-way branching program P' of size less than $2^{c(2m+1)/\log 2m}$ that computes m-GEN and such that $\max U(P') \leq 2 + c(2m+1)/\log 2m$. For m large enough, $2 + c(2m+1)/\log 2m \leq 3 + 2cm/\log m \leq 3cm/\log m$. Hence, by the first part of Proposition 5.1, for all $c > 0$ and infinitely many m there is a marking machine M that accepts all positive instances of m-GEN using at most $3cm/\log m$ markers. However, this is a contradiction to Proposition 5.3. ∎

5.2 Jumping automata on graphs

Graph s-t-connectivity is the major complete problem for the class of problems solvable in nondeterministic logarithmic space. We define it as follows.

> Problem STCONN
> *Given*: A directed graph G on vertices $\{1, \ldots, n\}$ with each vertex having out-degree two, where self-loops are allowed.
> *Determine*: Whether there is a path from vertex 1 to vertex n.

By n-STCONN we denote the sub-problem of STCONN restricted to graphs on n vertices. Given an instance G of n-STCONN we define an instance gen_G of $n+1$-GEN$_{(2rows)}$ in the following way: we let $gen_G(1,1) = 2$, $gen_G(2,1) = 1$ and furthermore if (i, j_1) and (i, j_2) are the two edges outgoing from vertex i in G, then we let $gen_G(1, i+1) = j_1 + 1$ and $gen_G(2, i+1) = j_2 + 1$. Clearly, 1 is connected to n in G iff $n+1 \in \langle\{1\}\rangle_{T^{gen_G}}$.

Extending the definition of Cook and Rackoff [CoRa80], Poon [Po93] defined *node-named jumping automata on graphs* (*NNJAG*) as a computational model to solve STCONN. A (deterministic) NNJAG J is a finite state automaton with p distinguished pebbles, q states and a transition function δ. (p, q and δ can non-uniformly depend on n.) The input to J is an instance G of STCONN. A configuration of J is the pair (s, Π) where s is the current state and Π is a mapping $\Pi : [p] \to [n]$ specifying the current position of each pebble on the input graph G. The initial configuration of J is a specific state s_0 with all pebbles placed on the vertex 1. When J is in the configuration

(s, Π) the transition function determines the next move of J based on the state s and the mapping Π. A move can be either a *walk* or a *jump*. A walk (r, i, s') consists of moving the pebble r along the i-th edge that comes out of the vertex $\Pi(r)$ in G and then assuming state s'. A jump (r, r', s') consists of moving pebble r to the vertex $\Pi(r')$ and assuming state s'. J accepts graph G if it ever enters an *accepting state* during its computation on G. NNJAG *solves* n-STCONN if on every instance G of n-STCONN, J accepts G iff there is a path from vertex 1 to vertex n in G. The *size of a NNJAG J* is the number of its possible configurations, i.e., qn^p.

We first establish the simulation lemma between NNJAG's and incremental branching programs.

Proposition 5.5 *1. For any deterministic syntactic incremental n^2-way branching program P that computes n^2-GEN$_{(2rows)}$ there is a NNJAG of size at most $5n^4 \cdot size(P)^2$ that solves n-STCONN.*

2. If J is an NNJAG solving n-STCONN then there is a deterministic syntactic incremental n-way branching program P of size at most $O(n^2 + n^3(size(J))^2)$ that computes n-GEN$_{(2rows)}$.

We present the proof in the Appendix.

There is a long sequence of lower bounds for various types of jumping automata on graphs. The strongest one was obtained by Edmonds, Poon and Achlioptas.

Proposition 5.6 *[EPA99] If NNJAG J solves n-STCONN then J has size at least $n^{\Omega(\log n)}$.*

As a corollary we obtain a lower bound for syntactic incremental branching programs.

Theorem 5.7 *There is a constant $c > 0$ such that for all n large enough if a deterministic syntactic incremental n-way branching program P solves n-GEN$_{(2rows)}$ then it has size at least $n^{c \log n}$.*

We should note here that Edmonds et al. in fact prove a lower bound for probabilistic NNJAG's. (A probabilistic NNJAG is defined as a probability distribution over deterministic NNJAG's where its size is the size of the largest NNJAG with a non-zero probability under that distribution. See [EPA99] for more details.) Their lower bound thus implies also a lower bound for appropriately defined probabilistic syntactic incremental branching programs.

6 Upper bounds

We present the proofs of the results of this section in the Appendix.

Theorem 6.1 *n-GEN can be computed by nondeterministic syntactic incremental branching programs and by $\{1\}$-monotone nondeterministic n-way branching programs of size $2^{n-2} + 1$.*

Theorem 6.2 *n-GEN can be computed by deterministic syntactic incremental branching programs of size $O(n^2 2^{n-2})$.*

Theorem 6.3 *n-GEN$_{(2rows)}$ can be computed by nondeterministic syntactic incremental branching programs and by $\{1\}$-monotone nondeterministic n-way branching programs of size $O(n^2)$.*

Together with Theorem 5.7, this gives a super-polynomial separation between the power of deterministic and nondeterministic syntactic incremental branching programs. The following theorem shows, that a stronger (e.g. exponential) separation is not possible.

Theorem 6.4 *If P is a nondeterministic syntactic incremental branching program that computes n-GEN then there is a deterministic syntactic incremental branching program P' of size at most $size(P)^{O(\log size(P))}$ that computes n-GEN.*

A similar relation between determinism and nondeterminism holds also for semantic incremental branching programs. We obtain as a corollary:

Theorem 6.5 *n-GEN$_{(2rows)}$ can be computed by deterministic syntactic incremental branching programs of size $n^{O(\log n)}$.*

Theorem 6.6 *n-GEN$_{(1row)}$ can be computed by deterministic syntactic incremental branching programs of size $(n-2)^2 + 3$.*

References

[BaMc91] D. Barrington and P. McKenzie, Oracle branching programs and Logspace versus P, *Information and Computation* **95** (1991), pp. 96–115.

[BJS01] P. Beame, T.S. Jayram and M.E. Saks, Time-Space Tradeoffs for Branching Programs J. Computer and Systems Science **63** (4), pp. 2001.542–572

[Be81] S.J. Berkowitz, *On some relationships between monotone and non-monotone circuit complexity*, manuscript, University of Toronto, Computer Science Department (1981).

[BoCo82] A. Borodin and S.A. Cook, A time-space trade-off for sorting on a general sequential model of computation *SIAM J. on Computing* **11**, 2 (1982), pp. 287–297.

[BRS93] A. Borodin, A. Razborov and R. Smolensky, On lower bounds for read-k-times branching programs. *Computational Complexity* **3** (1993) pp. 1-18.

[Co71] S.A. Cook, Characterizations of pushdown machines in terms of time-bounded computers, *J. of the Association for Computing Machinery* **18** (1), pp. 1971.4–18

[Co74] S.A. Cook, An observation on time-storage trade-off, J. Computer and Systems Science **9(3)** (1974), pp. 308–316.

[CoRa80] S.A. Cook and C.W. Rackoff, Space lower bounds for maze threadability on restricted machines, *SIAM J. on Computing* **9**, (1980), pp. 636–652.

[EPA99] J. Edmonds, C.K. Poon and D. Achlioptas, Tight lower bounds for st-connectivity on the NNJAG model, *SIAM J. on Computing* **28**, 6 (1999), pp. 2257–2284.

[JoLa77] N.D. Jones and W.T. Laaser, Complete problems for deterministic polynomial time, Theoretical Computer Science **3** (1977), pp. 105–117.

[Ju95] S. Jukna, A note on read-k-times branching programs, *RAIRO Theoretical Informatics and Applications* 29, pp. 75-83.

[KaWi88] M. Karchmer and A. Wigderson, Monotone circuits for connectivity require super-logarithmic depth, *Proc. of the 20th ACM Symp. on the Theory of Computing* (1988), pp. 539–550. Full version in: *SIAM J. on Disc. Math.* **3**, no. 2 (1990) pp. 255–265.

[PTC77] W. J. Paul, R. E. Tarjan and J. R. Celoni, Space bounds for a game on graphs, *Mathematical Systems Theory* **10**,(1977), 239-251.

[PaHe70] M.S. Paterson and C.E. Hewitt, Comparative schematology, *Record of Project MAC Conference on Concurrent Systems and Parallel Computations (June 1970)*, pp. 119-128, ACM, New Jersey, December 1970.

[Po93] C.K. Poon, Space bounds for graph connectivity problems on node-named JAGs and node-ordered JAGs *Proc. of the 34th IEEE Symp. on the Foundations of Computer Science* (1993), pp. 218–227.

[Ra91] A. Razborov, Lower bounds for deterministic and nondeterministic branching programs, *Proceedings of the 8th FCT, Lecture Notes in Computer Science*, **529** (1991) pp. 47-60.

[RaMc99] R. Raz and P. McKenzie, Separation of the monotone NC hierarchy, *Combinatorica* **19**(3)(1999), pp. 403-435.

[Sa70] W.J. Savitch, Relationships between nondeterministic and deterministic tape complexities, J. Computer and Systems Science **4(2)** (1970), pp. 177-192.

[We87] I. Wegener, The complexity of Boolean functions, Wiley-Teubner, 1987.

[We00] I. Wegener, Branching programs and binary decision diagrams, SIAM Monographs on Discrete Mathematics and Applications, 2000.

7 Appendix

Proof of Proposition 4.1

Let M be a monotone nondeterministic Boolean branching program computing f. For a given input $z \in \{0,1\}^t$ let G_z be the directed graph obtained from M by keeping those edges whose labels are satisfied by z. By definition, $f(z) = 1$ if and only if G_z has a directed path from the root to the ACCEPT node. This can be decided by monotone circuits in NC^2 using the adjacency matrix of the graph as input. The adjacency matrix of G_z can be determined from z as follows: for each pair of nodes (u,v) of the branching program M, we set the matrix entry (u,v) to 1 if the edge (u,v) is unlabeled, to 0 if there is no edge from u to v in M, and to z_i if the edge is labeled by $z_i = 1$ for some variable z_i. Since M is monotone, no other edge labels are possible. This gives a monotone circuit over the variables z_i computing f with depth $O((\log_2 s(t))^2)$.

Proof of Claim 4.11

First suppose that there is an edge (u,v) with label $i*j=n$ but its endpoint v is not the ACCEPT node. Since the program is incremental, we know that for any n-GEN instance g that can traverse this edge, $n \in \langle\{1\}\rangle_{T^g}$ must hold. Thus, we can redirect this edge such that its new endpoint is the ACCEPT node. Thus, we can assume without loss of generality, that all edges labeled $i*j=n$ have the ACCEPT node as their endpoint. Now suppose that there is an edge $(u, ACCEPT)$ with label $i*j=k$ for some $k \neq n$. If no n-GEN instance can traverse this edge, we can simply delete it from the program, or in the case of deterministic programs redirect it to a REJECT sink. Otherwise, if some n-GEN instance g traverses this edge, then $\mathrm{GEN}(g) = 1$ and thus $n \in \langle\{1\}\rangle_{T^g}$ must hold, and thus $n \in A_{\mathrm{GEN}}(u)$ must hold as well. But that is only possible if n appears in some edge label along a path from the root to u, which is a contradiction. To see that the above transformations preserve incrementality, note that for any given vertex u, we may have only removed some of its incoming edges but never added any new incoming edge (except for the ACCEPT or REJECT node). Thus the sets $A(u)$ and $A_{\mathrm{GEN}}(u)$ could only have increased.

Proof of Proposition 5.1

Both of the parts are easy to prove. For the first part. Let $u_1, u_2, \ldots, u_m$ be an accepting path of P on an instance T of n-GEN. Then inserting steps that remove markers when appropriate into the sequence $\emptyset, U(u_1), U(u_2), \ldots, U(u_m)$ we get an accepting computation of M on T that uses at most $\max U(P)$ markers. To see that the sequence obtained this way is indeed a legal sequence of steps for marking machines, it is enough to observe that in a syntactic incremental branching program, for any edge (u,v) with label $i*j=k$ $U(v) \setminus U(u) \subseteq \{k\}$ and also $\{i,j\} \subseteq U(u)$.

For the second part, we will construct a branching program P that simulates the computation of M. P will consists of $t = 1 + \sum_{i=1}^{\ell} \binom{n}{i}$ layers of nodes. For every $1 \le i < t$ and every non-empty set $S \subseteq [n]$ of size at most ℓ the i-th level will contain a node labeled (S,i). The t-th level will contain only the node ACCEPT. Let (S,i) be at level $1 \le i < t-1$. For each $p,q \in S \cup \{1\}$, $k \in [n]$ and $S' \subseteq S \cup \{k\}$, $|S'| \le \ell$, there is an edge from (S,i) leading to $(S', i+1)$ that is labeled by $p \star q = k$. Furthermore, from each node (S,t) there are edges leading to ACCEPT that are labeled by $p \star q = n$, for all $p,q \in S \cup \{1\}$. The initial node of P is $(\{1\},1)$. It is straightforward to verify that if M accepts all positive instances of GEN using at most ℓ markers then P computes n-GEN.

Proof of Proposition 5.5

For the proof of the first part, consider a deterministic syntactic incremental n^2-way branching program P that computes $n^2\text{-GEN}_{(2rows)}$. Using the technique from the proof of the Symmetrization Lemma we can build a deterministic syntactic incremental $n+1$-way branching program P' that computes $n+1\text{-GEN}_{(2rows)}$ whose size is at most $size(P)$ and for which $\max U(P') \le 2 + \log_n size(P)$. We will construct a NNJAG J solving n-STCONN with $p = 2 + \max U(P')$ pebbles and with the set of states consisting of nodes of P' (*main states*) with additional four *intermediate states* for each main state. NNJAG J on input G will simulate the computation of P' on gen_G. Whenever J will be in a main state u, the pebbles will be located at all vertices $J(u) = \{i-1;\ i \in U(u)$ and $i \ge 2\} \cup \{1\}$ so that the first pebble will be located at vertex 1. Notice $|J(u)| < p$. The initial state of J is the starting node of P' with all the pebbles placed at vertex 1. The accepting state of J is the ACCEPT node of P'. A step of computation of P' will be simulated in five steps of computation of machine J. Let u be a main state of J and let $i \star j = ?$ be the query asked by P' at node u. If $j > 1$ then J proceeds as follows.

1. In state u, let Π be the current mapping of the pebbles to vertices of G. Pick a pebble r such that there is another pebble r' with $\Pi(r') = \Pi(r)$. Jump pebble r to vertex 1, i.e., to the location of pebble 1, and assume state u_1.

2. In state u_1, jump pebble 1 to the location of pebble $\Pi^{-1}(j-1)$ and assume state u_2.

3. In state u_2, walk pebble 1 along the i-th edge leaving vertex $j-1$ and go to state u_3.

4. In state u_3, let Π' be the current mapping of the pebbles. Let v be the node of P that is reached from u via the edge labeled by $i * j = (\Pi')^{-1}(1) + 1$. While there is a pebble $r \neq 1$ such that $\Pi'(r) \notin J(v)$ jump r to vertex 1 and stay in state u_3. If there is no such a pebble do the following. If $\Pi'(1) \in J(v)$ then jump a pebble r, for which there is another pebble r' such that $\Pi'(r) = \Pi'(r')$, to the location of pebble 1 and assume state u_4. If $\Pi'(1) \notin J(v)$ assume state u_4 and do not move any pebble.

5. In state u_4, jump pebble 1 to vertex 1 and assume state v.

If $j = 1$ and $i = 1$, let v be the node of P' that is reached from u by the edge labeled $1 \star 1 = 2$. Then J in state u is going to move no pebbles and will just assume state v. Similarly, if $j = 1$ and $i = 2$ then J is going to move from state u to a state v that is reached from u via the edge labeled $2 \star 1 = 1$ in P'.

Clearly, all the moves described above are legal NNJAG moves that depend only on the current state of J and mapping of the pebbles. It is straightforward to verify that J on G simulates a computation of P' on gen_G, hence J accepts G iff P' accepts gen_G. As the size of J is $5 \cdot size(P') \cdot n^{2+\max U(P')} \leq 5n^4 \cdot size(P)^2$, J is the required NNJAG.

A statement similar to the second part of our claim is implicitly proven in [EPA99]. Edmonds et al. simulate NNJAG's by *pebble redundant* NNJAG's and then by branching programs. The resulting branching programs can be seen to be syntactic incremental under an appropriate notion of incrementality. In our context the proof is much simpler, since the argument in [EPA99] was aimed at proving a different statement. We present a direct proof below.

Let g be an instance of n-GEN$_{(2rows)}$. The syntactic incremental branching program P will have two parts. The first part will try to decide whether $n \in \langle\{1\}\rangle_{T^g}$, under the assumption that $2 \notin \langle\{1\}\rangle_{T^g}$. The second part of P will decide whether $n \in \langle\{1\}\rangle_{T^g}$ provided that $2 \in \langle\{1\}\rangle_{T^g}$.

If $2 \notin \langle\{1\}\rangle_{T^g}$, then g can be treated as an instance of n-GEN$_{(1row)}$ by considering only its first row. The first part of the program P that we construct, on an instance g of n-GEN$_{(2rows)}$ checks if either 2 or n is generated by just the first row of g. This can be done in size $O(n^2)$ as described in Theorem 6.6. Note also that $2 \in \langle\{1\}\rangle_{T^g}$ if and only if it is generated by just the first row. If $2 \notin \langle\{1\}\rangle_{T^g}$, then P rejects or accepts based on whether or not n is generated by just the first row. If $2 \in \langle\{1\}\rangle_{T^g}$, P will enter the starting node v_0 of the second part of P, i.e., all edges of the first part of P establishing that $2 \in \langle\{1\}\rangle_{T^g}$ will lead to the node v_0.

The second part of P will simulate the computation of J on a graph G, an instance of n-STCONN with the property that $n \in \langle\{1\}\rangle_{T^g}$ iff n is reachable from 1 in G. The graph G is determined by the instance g. The simulation is straightforward if J has the property that the graph of its configurations has no loops, and it works correctly also on every multi-graph of out-degree two, i.e., a graph with possible multiple edges between vertices (the two edges leaving a vertex may lead to the same vertex). We will argue below that we can modify J so that this is the case. Let us assume for now that J has the required property, i.e., it has no loops and works correctly on multi-graphs. We let G have vertices $\{1, \ldots, n\}$ and if $i \star 1 = j_1$ and $i \star 2 = j_2$ in g then we let the first edge from i in G lead to j_1 and the second edge lead to j_2. Clearly, G is of out-degree two and $n \in \langle\{1\}\rangle_{T^g}$ iff n is reachable from 1 in G.

The second part of the program P starting at v_0 is constructed as follows. The name of each node of P specifies some state of J together with some mapping of currently pebbled vertices of the input graph. This specifies a configuration of the NNJAG J. Since J is deterministic, this also specifies a move, which can either be a jump or a walk. In case it is a walk, it involves a specific vertex i of G and one of the edges leaving the given vertex. Since we consider only graphs of outdegree two, in our case this will give a pair of the form either $i * 1$ or $i * 2$. We will think of this pair as the question associated with the given node of the program P, thus the n edges leaving the given node will be labeled $i * 1 = k$ for $k = 1, \ldots, n$ (or $i * 2 = k$ for $k = 1, \ldots, n$, respectively). For each edge there is exactly one node (that is, a pebble location/state pair) which corresponds to the configuration of J following the corresponding move, and we use that node as the endpoint of the given edge. If the move of J corresponding to a given node of P is a jump, we put in n edges labeled $1 * 1 = k$ for $k = 1, \ldots, n$ all leading to the same node of P corresponding to the configuration of J after the jump. It is easy to see that the program constructed this way is syntactic incremental, since the moves of J only involve vertices of its input graph that are already pebbled. It remains to modify P so that all edges leading to an accepting configuration of J will lead to the node ACCEPT, and edges to rejecting configurations will lead to REJECT. The program constructed this way will always correspond to simulating J on a (multi-)graph G of out-degree two, and since $n \in \langle\{1\}\rangle_{T^g}$ if and only if there is a path from 1 to n in G, the program will be correct.

It remains to argue that we can modify J so that it will compute correctly on multi-graphs and its configuration graph will have no loops. This modification increases the size of J to at most $O(n^3(size(J))^2)$.

It is standard to eliminate loops of the computation by at most squaring the number of configurations: this can be achieved by adding a time step counter to the name of each configuration. To deal with multiple edges, we note that for any graph G of out-degree two where at least one of the edges from the vertex 1 leads to another vertex, the following graph G' has no multiple edges, and n is reachable from 1 in G' if and only if n is reachable from 1 in G. Let the first edge from i lead to j_1 and the second edge from i lead to j_2 in G. If $j_1 \neq j_2$, we put in the same two edges for i in G'. If $j_1 = j_2 \neq 1$ then we let the first edge from i in G' lead to j_1 and the second edge lead to 1. Otherwise $j_1 = j_2 = 1$ and we let the first edge of G' to lead to vertex 1 and the second one will form a self loop at i. Clearly, G' is of out-degree two and has the desired properties.

We modify J so that it would compute on a graph G as if it were computing on the corresponding graph G'. We equip J with three additional auxiliary pebbles. We will leave one of them at vertex 1 during the whole computation. Using one of the auxiliary pebbles we can first verify that at least one edge from the vertex 1 leads to another vertex in G. If not J immediately rejects (assuming $1 \neq n$). Otherwise J computes as before with the following modifications.

For each configuration where the next move of J on G is a walk involving the second edge out of some vertex i we invoke the following subprogram. We jump two auxiliary pebbles to the vertex i and move one of these pebbles along the first edge from i and the other one along the second edge from i. If the two auxiliary pebbles do not coincide we move the pebble r that J would move originally along the second edge. If the two auxiliary pebbles coincide but they are not at vertex 1 we jump the pebble r to the vertex 1, i.e., the location of the third auxiliary pebble. Otherwise all the three auxiliary pebbles coincide and we keep pebble r at vertex i. In all three cases J assumes the state that it would originally assume. It is clear from the construction that the modified J will compute on G as if J would compute on G'. Also, our modification multiplies the number of states of J by a factor of at most five and increases the number of pebbles by three, hence it increases the size of J by only the allowed amount.

Proof of Theorem 6.1

For each $S \subseteq \{2, \ldots, n-1\}$ we create a node labeled by S. The program will consist of these 2^{n-2} nodes and a sink node labeled ACCEPT. The node labeled by S will have an outgoing edge labeled $i * j = k$ for each triple such that $\{i, j\} \subseteq S \cup \{1\}$ and $k \notin S \cup \{1\}$. (For $S \neq \emptyset$ the edges labeled $1 * 1 = k$ can be omitted.) Each edge with label $i * j = n$ goes to ACCEPT. An edge labeled $i * j = k$ out of the node labeled by S goes to the node labeled $S \cup \{k\}$.

We did not use labels of the form $i * j = 1$, thus the program is $\{1\}$-monotone. Let us denote by u_S the node with label S. We have $A(u_S) = S \cup \{1\}$ for each S, thus the program is syntactic incremental.

Proof of Theorem 6.2

The previous construction can be modified to obtain a deterministic program as follows. Replace the node labeled S by a group of nodes labeled $< S, i, j >$, for pairs $\{i, j\} \subseteq S \cup \{1\}$ (omitting the pair $(1, 1)$ for $S \neq \emptyset$). Order the nodes within each group in an arbitrary way. The group corresponding to the empty set consists of a single node labeled $< \emptyset, 1, 1 >$, and this node will be the root. Each edge with label $i * j = n$ goes to ACCEPT. Out of the node with label $< S, i, j >$ the edges labeled $i * j = k$ where $k \notin S \cup \{1\}$ go to the first node of the group corresponding to $S \cup \{k\}$. For all but the last node of each group, send the edges with label $i * j = k$ to the next node of the same group if $k \in S \cup \{1\}$. For the last node of each group, the edges labeled $i * j = k$ with $k \in S \cup \{1\}$ go to REJECT. For each node u whose label starts with S we have $A(u) = S \cup \{1\}$, thus the program is syntactic incremental.

Proof of Theorem 6.3

The program has n layers of nodes. The first and last layer consists of a single node: the root on the first layer, and the ACCEPT node on the last layer. The second layer has $n-2$ nodes called t_2 and $w_{(2,3)}, \ldots, w_{(2,n-1)}$. Layers $l = 3, \ldots, n-1$ have $2n-5$ nodes each, called t_l, $w_{(l,3)}, \ldots, w_{(l,n-1)}$ and $u_{(l,3)}, \ldots, u_{(l,n-1)}$.

There are $n-1$ edges leaving the root labeled $1 * 1 = k$ going to nodes $w_{(2,k)}$ for $k = 3, \ldots, n-1$, the edge $1 * 1 = 2$ goes to t_2, and $1 * 1 = n$ goes to ACCEPT. The nodes $w_{(l,j)}$ for $l \leq n-2$ have $n-1$ edges labeled $1 * j = k$ going to nodes $w_{(l+1,k)}$ for $k = 3, \ldots, n-1$, the edge $1 * j = 2$ goes to $t_{(l+1)}$, and $1 * j = n$ goes to ACCEPT.

There are $3(n-1)$ edges leaving each node t_l for $l \leq n-2$, labeled $1 * 2 = k$, $2 * 1 = k$ and $2 * 2 = k$ going to $t_{(l+1)}$ if $k = 2$ and going to the node $u_{(l+1,k)}$ for $k = 3, \ldots, n-1$, and to the ACCEPT node if $k = n$.

For $l = 3, \ldots, n-2$, and $j = 3, \ldots, n-1$, there will be $2(n-1)$ edges leaving the node $u_{(l,j)}$, labeled $1 * j = k$ and $2 * j = k$ for $k = 2, \ldots, n$. The edges with labels $1 * j = k$ and $2 * j = k$ go to the node $u_{(l+1,k)}$ for $k = 3, \ldots, n-1$, and to $t_{(l+1)}$ if $k = 2$. All edges labeled $i * j = n$ go to ACCEPT. For the last layer, each node $u_{(n-1,j)}$ and $w_{(n-1,j)}$ has only two outgoing edges labeled $1 * j = n$ and $2 * j = n$ going to ACCEPT. The node $t_{(n-1)}$ has 3 outgoing edges labeled $1 * 2 = n$, $2 * 1 = n$ and $2 * 2 = n$ going to ACCEPT. The program is clearly syntactic incremental. Since we never use labels of the form $i * j = 1$, the program is also $\{1\}$-monotone.

It remains to prove that the program is correct. To see this, we represent each n-GEN$_{(2rows)}$ instance g by a graph G_g with n nodes. There will be an edge going from node j to node k with label $1 * j = k$ $(2 * j = k)$ if

$(1, j, k) \in T^g$ ($(2, j, k) \in T^g$, respectively). There is a path from the node 1 to the node n in G_g if and only if $n \in \langle\{1\}\rangle_T$. Moreover, if there is at least one path from 1 to n, then a path from 1 to n of length at most n visiting each node at most once must also exist. Thus, for any positive instance g, such a path can be found in G_g. But notice that for any path π of G_g from 1 to n visiting each node i at most once the program we constructed must contain a path from the root to the ACCEPT node with the exact same sequence of edge labels as π, and g can follow this path in the branching program. Thus our program accepts every positive instance g. On the other hand, since the program is syntactic incremental and all edges adjacent to the ACCEPT node have labels of the form $i * j = n$, the program must reject every negative instance g.

Proof of Theorem 6.4

Let P be a nondeterministic syntactic incremental branching program that computes n-GEN. By standard techniques we can convert it into an equivalent nondeterministic syntactic incremental branching program of size $m \in O(n \cdot (size(P))^2)$ computing n-GEN that has at most two edges outgoing from every node. Hence, we assume P has this form. We will view P as a graph. Let g be an instance of n-GEN. Deciding whether P accepts g is the same as deciding whether there is a path from the initial node to the accepting node of P in which all the edges that are labeled inconsistently with g are removed.

Cook and Rackoff's results [CoRa80] imply that there is a deterministic NNJAG J that has $O(m^4)$ internal states and uses $O(\log m)$ pebbles and that on any directed graph G of size m with all vertices of out-degree at most two computes so that every vertex in G that is reachable by a path from vertex 1 contains a pebble (*is pebbled*) at some point during the computation. (Here the definition of behavior of NNJAG's is extended so that if an NNJAG requests a walk of a pebble along a non-existent edge then the pebble stays at its current location and only the internal state of the NNJAG changes.) From properties of NNJAG's it follows that if a vertex u of G is pebbled during a computation of an NNJAG on G then there is a path in G from vertex 1 to u so that some pebble is walked along every edge of this path at some point of the computation prior to pebbling u.

Our deterministic syntactic incremental branching program P' on an input instance g of n-GEN will simulate J on P in which all edges that are inconsistent with the input instance g are removed. In the names of its nodes P' will record the internal state of J, positions of its pebbles on P and the number of steps it computed so far. Let J have q internal states and use p pebbles. The set of nodes of P' will be $[qm^p + 1] \times [q] \times [m]^p \cup \{\mathrm{ACCEPT}, \mathrm{REJECT}\}$. Let v_0 be the initial node of P and s_0 be the initial state of J. Node $(1, s_0, v_0, v_0, \ldots, v_0)$ is the initial node of P'. ACCEPT is the accepting node of P'.

Let edges of P' be defined as follows. Let $1 \le t \le qm^p$. Let J in state s with pebbles located at $u_1, \ldots, u_p$ walk a pebble r along d-th outgoing edge from node u_r and assume state s'. Assume that the d-th outgoing edge from node u_r in P is labeled by $i \star j = k$, and that it leads to a node v. If $v \neq$ACCEPT then there is an edge labeled $i \star j = k$ in P' leading from node $(t, s, u_1, \ldots, u_p)$ to node $(t + 1, s', u_1, \ldots, u_{r-1}, v, u_{r+1}, \ldots, u_p)$ and for all $k' \in [n] \setminus \{k\}$, there are edges labeled $i \star j = k'$ from $(t, s, u_1, \ldots, u_p)$ to $(t + 1, s', u_1, \ldots, u_p)$. If $v =$ACCEPT then let there be an edge labeled $i \star j = k$ leading from $(t, s, u_1, \ldots, u_p)$ to ACCEPT and let for all $k' \in [n] \setminus \{k\}$, there be edges labeled $i \star j = k'$ from $(t, s, u_1, \ldots, u_p)$ to $(t + 1, s', u_1, \ldots, u_p)$. Assume now that the d-th outgoing edge from node u_r in P is unlabeled and that it leads to a node v. Then for all $k \in [n]$, there will be an edge labeled by $1 \star 1 = k$ going from $(t, s, u_1, \ldots, u_p)$ to node U where $U = (t + 1, s', u_1, \ldots, u_{r-1}, v, u_{r+1}, \ldots, u_p)$ if $v \neq ACCEPT$, and $U = ACCEPT$ otherwise. If there is no d-th outgoing edge from u_r then for all $k \in [n]$, there will be an edge labeled by $1 \star 1 = k$ going from $(t, s, u_1, \ldots, u_p)$ to node $(t + 1, s', u_1, \ldots, u_p)$. If J in state s with pebbles located at nodes $u_1, \ldots, u_p$ jumps a pebble r to pebble r' and assumes state s' then for all $k \in [n]$, let there be an edge labeled $1 \star 1 = k$ leading from $(t, s, u_1, \ldots, u_p)$ to $(t + 1, s', u_1, \ldots, u_{r-1}, u_{r'}, u_{r+1}, \ldots, u_p)$. All other edges in P' lead to REJECT.

Clearly, the size of P' is at most $q^2 m^{2p} + 2 \le size(P)^{O(\log size(P))}$ as $n \le size(P)$. The fact that P' computes n-GEN follows from the fact that P computes n-GEN and J traverses all nodes of P reachable on an input instance of n-GEN.

It remains to argue that P' is syntactic incremental. To see that let us consider a query "$i \star j =?$" asked at some node $(t, s, u_1, \ldots, u_p)$. Let π be an arbitrary path from the initial node of P' to $(t, s, u_1, \ldots, u_p)$. Because of the properties of J and the construction of P' there must be paths $\pi_1, \ldots, \pi_p$ in P that lead from the initial vertex of P to nodes $u_1, \ldots, u_p$ so that P' asks all the queries that appear as labels along paths $\pi_1, \ldots, \pi_p$. Since the query "$i \star j =?$" is indeed a query of P at one of the nodes $u_1, \ldots, u_p$ and P is syntactic incremental, both i and j must be in $\langle\{1\}\rangle_{T^{\pi_1}} \cup \cdots \cup \langle\{1\}\rangle_{T^{\pi_p}}$. Hence, it also must be in $\langle\{1\}\rangle_{T^{\pi}}$. Thus, P' is syntactic incremental.

Note that the above construction is similar to the simulation of NNJAG's by branching programs in [EPA99], but it involves some additional ideas to obtain the relationship between deterministic and nondeterministic syntactic incremental branching programs.

Proof of Theorem 6.5

This follows from the proof of the previous theorem and Theorem 6.3.

Proof of Theorem 6.6

The construction is similar to the construction in the proof of Theorem 6.3, in fact in some sense it is a subprogram of it. Now we have $(n-2)$ layers of $(n-2)$ nodes each for $l, k = 2, \ldots, n-1$, in addition to the root $u_{(1,1)}$, an ACCEPT and a REJECT node. Out of each non-sink node $u_{(l,j)}$, we have exactly n outgoing edges, labeled $1 * j = k$, for $k = 1, \ldots, n$. If $l \leq n-2$, the edges for $k = 2, \ldots, n-1$ go to $u_{(l+1,k)}$ (at the next layer). All edges with label $1 * j = k$ for $k \neq n$ from the last layer go to REJECT, and all edges with label $1 * j = 1$ (from any layer) go to REJECT. All edges with label $1 * j = n$ (from any layer) go to ACCEPT.

7.1 Cook vs Paterson and Hewitt rules

In this section we discuss the difference between marking machines that follow the marking rules of Cook [Co74] and those that follow the pebbling rules of Paterson and Hewitt [PaHe70]. There are three types of moves of a marking machine: 1. mark the source element 1; 2. remove a mark from some marked element; 3. the inductive move. The definition of Cook differs from the definition of Paterson and Hewitt in the inductive move. We use the inductive move of Paterson and Hewitt which on a relation T may mark a new element z of the universe if there exist two marked elements x and y such that $(x, y, z) \in T$. The inductive move of Cook also marks the new element z if there are two marked elements x and y such that $(x, y, z) \in T$ but at the same time it removes a mark from either x or y. Hence the inductive move of Cook corresponds to moving the mark from either x or y to z.

We claim that the rules of Paterson and Hewitt [PaHe70] result in more efficient marking machines than the ones defined by Cook [Co74]. We call the marking machines that follow the rules of Paterson and Hewitt *fat*, and we call the machines that follow the rules of Cook *slim*. For every $n \geq 2$ we present here a relation T_n on $[2n]$ such that the target element $2n$ can be marked by a fat marking machine using only 4 marks whereas slim marking machines need at least n marks.

For $n \geq 2$, let T_n be the ternary relation on $[2n]$ which contains the following elements: $(1, 1, 2), (1, 1, 3)$ and $(2n-2, 2n-1, 2n)$ are in T_n and for every $1 \leq i < n-1$, $(2i, 2i+1, 2i+2)$ and $(2i, 2i+1, 2i+3)$ are in T_n. (This relation corresponds to a layered graph with $n+1$ levels each of size at most two, where edges go from one level to the next.) If is straightforward to verify that a fat marking machine needs only four marks to mark the target element $2n$. However, we claim that a slim marking machine requires at least n marks to mark $2n$.

Claim 7.1 *For $n \geq 2$, on T_n a slim marking machine needs at least n marks to mark the element $2n$.*

Proof. Let $n \geq 2$ and let $C_0, C_2, \ldots, C_t$ be an accepting computation of a slim marking machine on T_n. Let C be an arbitrary configuration of this computation. Let $1 \leq m \leq n$. We say that *layers $m, \ldots, n$ are full in C* if at least $n+1-m$ elements among $\{2m, 2m+1, \ldots, 2n\}$ are marked in C. Clearly, layers $n, \ldots, n$ are full (is full) in C_t. Let $1 \leq m$ be the smallest integer such that layers $m, \ldots, n$ are full in some configuration of the accepting computation. We claim that $m = 1$.

For a contradiction assume that $m > 1$. The only way how the number of marked elements among $\{2m, 2m+1, \ldots, 2n\}$ can increase is by moving there a mark from elements $\{2m-2, 2m-1\}$. Let C_l be the first configuration when layers $m, \ldots, n$ are full. Then in C_{l-1} both elements $2m-2$ and $2m-1$ are marked. But that means that layers $m-1, m, \ldots, n$ are full in a contradiction to the minimality of m. The statement now follows. ∎