Proof complexity of systems of (non-deterministic) decision trees and branching programs

Sam Buss
Dept. of Mathematics, UC San Diego
sbuss@ucsd.edu

Anupam Das
Dept. of Computer Science, University of Copenhagen
anupam.das@di.ku.dk

Alexander Knop
Dept. of Mathematics, UC San Diego
aknop@ucsd.edu

Abstract

This paper studies propositional proof systems in which lines are sequents of decision trees or branching programs, deterministic or non-deterministic. Decision trees (DTs) are represented by a natural term syntax, inducing the system LDT, and non-determinism is modelled by including disjunction, $\vee$, as primitive (system LNDT). Branching programs generalise DTs to dag-like structures and are duly handled by extension variables in our setting, as is common in proof complexity (systems eLDT and eLNDT).

Deterministic and non-deterministic branching programs are natural nonuniform analogues of log-space (L) and nondeterministic log-space (NL), respectively. Thus eLDT and eLNDT serve as natural systems of reasoning corresponding to L and NL, respectively.

The main results of the paper are simulation and non-simulation results for tree-like and dag-like proofs in LDT, LNDT, eLDT and eLNDT. We also compare them with Frege systems, constant-depth Frege systems and extended Frege systems.

2012 ACM Subject Classification Theory of computation → Computational complexity and cryptography

Keywords and phrases proof complexity, decision trees, branching programs, logspace, sequent calculus, non-determinism, low-depth complexity

Digital Object Identifier 10.4230/LIPIcs.CSL.2020.12

Funding *Sam Buss*: This work supported in part by Simons Foundation grant 578919
Anupam Das: This work was supported by a Marie Skłodowska-Curie fellowship, *Monotonicity in Logic and Complexity*, ERC project 753431.

1 Introduction

Propositional proof systems are widely studied because of their connections to feasible complexity classes and their usefulness for computer-based reasoning. The first connections to computational complexity arose largely from the work of Cook and Reckhow [11, 16, 17], showing a connection to the NP-coNP question. These results, building on the work of Tseitin [33] initiated the study of the relative efficiency of propositional proof systems. The present paper is introduces propositional proof systems that are closely connected to log-space (L) and nondeterministic log-space (NL).

Our original motivation for this study was to investigate propositional proof systems corresponding to the first-order bounded arithmetic theories VL and VNL for L and NL, see [15]. This follows a long line of work defining formal theories of bounded arithmetic that correspond to computational complexity classes, as well as to provability in propositional


© Sam Buss, Anupam Das, and Alexander Knop;
licensed under Creative Commons License CC-BY

proof systems. The first results of this type were due (independently) to Paris and Wilkie [30] who gave a translation from $I\Delta_0$ to constant-depth Frege (AC^0-Frege) proofs and to Cook [11] who gave a translation from PV to extended Frege (e$\mathcal{F}$) proofs. Since the first-order bounded arithmetic theory S^1_2 is conservative over the equational theory PV, Cook's translation also applies to the bounded arithmetic theory S^1_2 [7]. As shown in the table below, similar propositional translations have since been given for a range of other theories, including first-order, second-order and equational theories.

Formal Theories	Propositional Proof Systems	Complexity Class	
PV, S^1_2	e$\mathcal{F}$	P	[11, 7]
PSA, U^1_2	QBF	PSPACE	[18, 7]
T^i_2, S^{i+1}_2	G_i, G^*_{i+1}	$P^{\Sigma^p_i}$	[27, 28, 7]
VNC^0	Frege ($\mathcal{F}$)	ALogTime	[14, 15, 1]
VL	GL*	L	[31, 15]
VNL	GNL*	NL	[32, 15]

For an introduction to these and related results, see the books [7, 15, 25, 26]. A hallmark of the table above is that the lines in the propositional proofs express (nonuniform) properties in the corresponding complexity class. For instance, lines in a Frege proof are propositional formulas, for which the evaluation problem is complete for alternating log-time (ALogTime), cf. [8]. Likewise, lines in an e$\mathcal{F}$ proof are (implicitly) Boolean circuits, for which the evaluation problem is complete for P, cf. [29].

This paper's main goal is to define alternatives for the proof systems GL* and GNL* corresponding to log-space and nondeterministic log-space (see [31, 32, 12, 13]). GL* restricts cut formulas to be "ΣCNF(2)" formulas; the subformula property then implies that proofs contain only ΣCNF(2) formulas when proving ΣCNF(2) theorems. GNL* similarly restricts cut formulas to be "ΣKrom" formulas.[1] ΣCNF(2) and ΣKrom do have expressive power equivalent to nonuniform L and NL respectively [22, 19], but they are are somewhat ad hoc classes of quantified formulas, and their connections to L and NL are indirect. In this paper, we propose new proof systems, eLDT and eLNDT, as alternatives for GL* and GNL* respectively. The lines in eLDT and eLNDT proofs are sequents of formulas expressing *branching programs* and *nondeterministic branching programs*, respectively. This follows an earlier unpublished suggestion of S. Cook [10], who gave a system for L based on branching programs via "Prover-Liar" games (see [9]). The advantage of our systems is that deterministic and nondeterministic branching programs correspond directly to nonuniform L and NL respectively and do not require the use of quantified formulas. (See [34] for a comprehensive introduction to branching programs.)

To design the proof systems eLDT and eLNDT, we need to choose representations for branching programs. For this, we use a formula-based representation, as this fits well into the customary frameworks for proof systems. Since formulas only represent tree-structures, we first define the systems LDT and LNDT for decision trees and non-deterministic decision trees, respectively. From here dag-like structures are described using extension variables,

[1] A ΣKrom formula has the form if it has the form $\exists\vec{z}\phi(\vec{z},\vec{x})$, where ϕ is a conjunction $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ with each C_i a disjunction of any number of x-literals and at most two z-literals.

allowing us to abbreviate complex formulas by fresh variables, yielding the systems eLDT and eLNDT. An example this is given in Figure 2 on page 12. This is similar to the way the extension variables in extended Frege proofs allow circuits to be expressed by small formulas.

We start in Section 2 describing proof systems LDT and LNDT that work with just deterministic and nondeterministic decision trees (without extension variables). Deterministic decision trees are represented by formulas using a single "case" or 'if-then-else" connective, written in infix notation ApB, which means "if p is false, then A, else B". The condition p is required to be a literal, but A and B are arbitrary formulas. The system LDT is a sequent calculus system in which all formulas are decision trees. Nondeterministic decision trees may further be composed by disjunctions, allowing formulas of the form $(A \vee B)$. The system LNDT is a sequent calculus in which all formulas are nondeterministic decision trees. LDT and LNDT are weak systems; in fact, they are both polynomially simulated by depth-2 LK that is, by the sequent calculus LK with all formulas are depth two, allowing proofs to be dag-like. Figure 1 shows the equivalences between systems as currently established; those that concern LDT and LNDT are given in Section 4. Section 5 introduces the proof systems eLDT and eLNDT for branching programs and nondeterministic branching programs.

One issue in designing these proof systems is the treatment of isomorphic or bisimilar branching programs. One approach is to allow proofs to freely replace any branching program with any isomorphic or bisimilar branching program by means of additional axioms, e.g. as done by Jeřábek [21] for the reformulation of extended Frege using Boolean circuits as lines. The problem with using isomorphism or bisimilarity axioms is that these problems (for branching programs) are in NL but not known to be in L. Such axioms are thus undesirable, at least for eLDT, as it is a proof system for log-space. We instead adopt a more conservative approach: the equivalence of bisimilar branching programs must be proved explicitly.

Since formulas in eLDT and eLNDT proofs express nonuniform L and NL properties, respectively, they are intermediate in expressive power between Boolean formulas (expressing NC^1 properties) and Boolean circuits (expressing nonuniform P properties). Thus it is not surprising that, as shown in Figure 1, these two systems are between Frege and extended Frege in strength. In addition, since NL properties can be expressed by quasipolynomial size formulas, it is not unexpected that Frege proofs can quasipolynomially simulate eLNDT, and hence eLDT. These results are given in Section 6.

We include only brief proof sketches in this paper, due to space constraints, but full proofs may be found in [5].

2 Decision tree formulas and LDT proofs

This section describes decision tree (DT) formulas, and the associated sequent calculus proof system LDT. All our proof systems are *propositional* proof systems with variables $x, y, z \ldots$ intended to range over the Boolean values *False* and *True*. We use 0 and 1 to denote the constants *False* and *True*, respectively. A *literal* is either a propositional variable x or a negated propositional variable $\overline{x}$. We use use variables $p, q, r, \ldots$ to range over literals.

The only connective for forming decision tree formulas (DT formulas) is the 3-ary "case" function, written in infix notation as (ApB) where A and B are formulas and p is required to be a literal. This informally means "if p is false, then A, else B". Formally:

▸ **Definition 1.** Decision tree formulas*, or* DT formulas*, are inductively defined as follows:*

1. *any literal p is a* DT *formula, and*
2. *if A and B are* DT *formulas and p is a literal, then (ApB) is a* DT *formula. We call p the* decision literal *of this formula.*

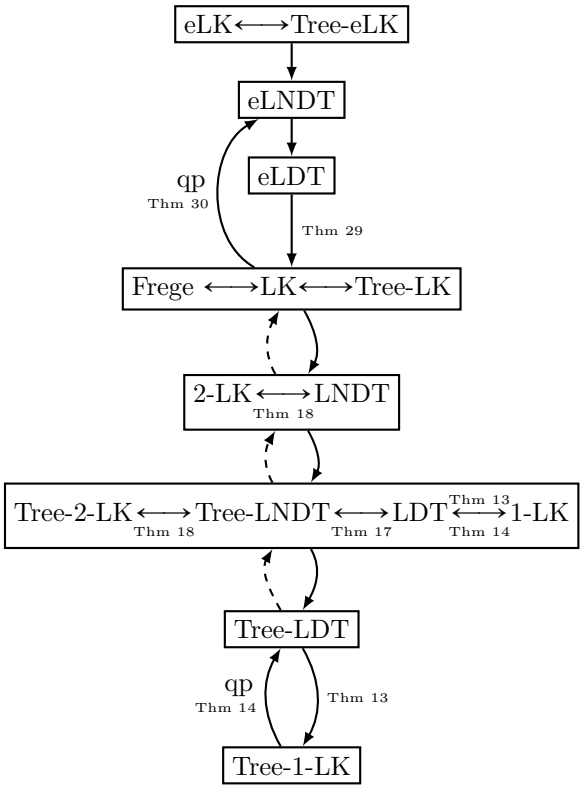


Figure 1 Relations between proof systems. $\to$ means 'polynomially simulates'; $\to_{qp}$ means 'quasipolynomially simulates'; $\dashrightarrow$ means 'exponentially separated from'. d-LK is the system of dag-like LK proofs with only depth d formulae occurring (atomic formulae have depth 0) By default, all proof systems allow dag-like proofs, unless they are labeled as "Tree".

The size *of a* DT *formula A is the number of occurrences of atomic formulas in A.*

Suppose α is a 0-1-truth assignment to the variables; the semantics of DT formulas is defined by extending α to be a truth assignment to all DT formulas by inductively defining:

$$\alpha(\overline{x}) = 1 - \alpha(x) \tag{1}$$

$$\alpha(ApB) = \begin{cases} \alpha(A) & \text{if } \alpha(p) = 0 \\ \alpha(B) & \text{otherwise.} \end{cases}$$

It is important that only *literals* p serve as the case distinctions in DT formulas. Notably, for C a complex formula, an expression $(A\,C\,B)$, which evaluates to A if C is true and to B if C is false, would in general denote a decision *diagram* rather than a decision tree.

Although there is no explicit negation of DT formulas, we informally define the negation $\overline{A}$ of a DT formula inductively by letting $\overline{\overline{x}}$ denote x, and letting $\overline{ApB}$ denote the formula $\overline{A}\,p\,\overline{B}$. Of course $\overline{A}$ is a DT formula whenever A is, and $\overline{A}$ correctly expresses the negation of A. Notice also that negative decision literals are 'syntactic sugar', since $Ap\bar{p}B$ is equivalent to BpA. Nonetheless the notation is useful for making later definitions more intuitive.

Our definition of DT formulas is somewhat different from the usual definition of decision trees. The more common definition would allow 0 and 1 as atomic formulas instead of literals p as in condition 1 of Definition 1. We call such formulas 0/1-DT formulas; they are

equivalent to DT formulas in expressive power. The constants 0 and 1 are equivalent to $pp\overline{p}$ and $\overline{p}pp$, for any literal p. More generally, $0pA$, $1pA$, $Ap0$ or $Ap1$ are equivalent to ppA, $\overline{p}pA$, $Ap\overline{p}$, or App, respectively. Conversely, a literal p, when used as atom, is equivalent to $0p1$.

▸ Remark 2 (Expressive power of decision trees). It is easy to decide the validity or satisfiability of a DT formula with a log-space algorithm. To check satisfiability, for example, one examines each leaf in the formula tree (each atomic subformula p) and verifies whether the path of literals from the root to the leaf is consistent with some truth assignment.

A DT formula A of size n can be expressed as a DNF formula of size $O(n^2)$ with at most n disjuncts, defined formally in Section 3. Informally this DNF is formed by taking the disjunction of terms (a.k.a conjunctions of literals) corresponding to paths from the root to a leaf. A dual construction expresses a DT formula A as a CNF formula of size $O(n^2)$ with at most n conjuncts. It is folklore that the construction can be partially reversed: namely any Boolean function that is equivalently expressed by a DNF φ and a CNF ψ can be represented by a DT formula of size quasipolynomial in the sizes of φ and ψ. This bound is optimal, as [23] proves a quasipolynomial lower bound.

We next define the proof system LDT for reasoning about DT formulas. Lines in an LDT proof are sequents, hence they express disjunctions of DT's. Thus lines in LDT proofs can express DNF properties, whose validity problem is non-trivial, indeed coNP-complete.

▸ **Definition 3.** *A* cedent*, denoted Γ, Δ etc., is a multiset of formulas; we often use commas for multiset union, and write Γ, A for the multiset $\Gamma, \{A\}$. A* sequent *is an expression $\Gamma \to \Delta$ where Γ and Δ are cedents. Γ and Δ are called the* antecedent *and* succedent*, respectively.*

The intended meaning of $\Gamma \to \Delta$ is that if every formula in Γ is true, then some formula in Δ is true. Accordingly, $\Gamma \to \Delta$ is true under a truth assignment α iff $\alpha(A) = 0$ for some $A \in \Gamma$ or $\alpha(A) = 1$ for some $A \in \Delta$. A sequent is *valid* iff it is true for every truth assignment.

▸ **Definition 4.** *The sequent calculus* LDT *is a proof system in which lines are sequents of* DT *formulas. The valid initial sequents (axioms) are, for p any literal,*

$$p \to p \qquad p, \overline{p} \to \qquad \to p, \overline{p}.$$

The rules of inference are:

Contraction rules:

$$\text{c-l: } \frac{A, A, \Gamma \to \Delta}{A, \Gamma \to \Delta} \qquad \text{c-r: } \frac{\Gamma \to \Delta, A, A}{\Gamma \to \Delta, A}$$

Weakening rules:

$$\text{w-l: } \frac{\Gamma \to \Delta}{A, \Gamma \to \Delta} \qquad \text{w-r: } \frac{\Gamma \to \Delta}{\Gamma \to \Delta, A}$$

Cut rule:

$$\text{cut: } \frac{\Gamma \to \Delta, A \qquad A, \Gamma \to \Delta}{\Gamma \to \Delta}$$

Decision rules:

$$\text{dec-l: } \frac{\Gamma, A \to p, \Delta \qquad \Gamma, p, B \to \Delta}{\Gamma, ApB \to \Delta}$$

$$\text{dec-r: } \frac{\Gamma \to A, p, \Delta \qquad \Gamma, p \to B, \Delta}{\Gamma \to ApB, \Delta}$$

Proofs are, by default, dag-like. I.e. a proof *of a sequent S in* LDT *is a sequence $(S_0, \ldots, S_n)$ such that S is S_n and each S_k is either an initial sequent or is the conclusion of an inference step whose premises occur amongst $(S_i)_{i<k}$. The subsystem where proofs are restricted to be tree-like (i.e. trees of sequents composed by inference steps) is denoted* Tree-LDT*.*

The size *of a proof is the sum of the sizes of the formula occurrences in the proof.*

The inference rules that are new to LDT are the two decision rules, *dec-l* and *dec-r*. Since ApB is equivalent to $(A \vee p) \wedge (\overline{p} \vee B)$, the lower sequent of a *dec-r* is true (under some fixed truth assignment) iff both upper sequents are true under the same assignment, i.e. the rule is sound and invertible. Similarly, since ApB is also equivalent to $(A \wedge \overline{p}) \vee (p \wedge B)$, the *dec-l* rule is also sound and invertible.

▸ Remark 5 (Cut-free completeness). The invertibility properties also imply that the cut-free fragment of LDT is complete. To prove this by induction on the complexity of sequents, start with a valid sequent $\Gamma \to \Delta$; choose any non-atomic formula ApB in Γ or Δ, and apply the appropriate decision rule *dec-l* or *dec-r* that introduces this formula. The upper sequents of this inference are also valid and, furthermore, they have logical complexity strictly less than the logical complexity of $\Gamma \to \Delta$. The base case of the induction is when $\Gamma \to \Delta$ contains only atomic formulas; in this case, it can be inferred from an initial sequent with weakenings. Note that this shows in fact, that any valid sequent can be proved in LDT using only decision rules, weakenings, and initial sequents. The system also enjoys a 'local' cut-elimination procedure, via standard techniques, but that is beyond the scope of this work.

▸ **Proposition 6.** *The following have polynomial size, cut-free,* Tree-LDT *proofs:*

(a) $A \to A$ (b) $\to A, \overline{A}$ (c) $A, \overline{A} \to$ (d) $A \to p, ApB$ (e) $p, B \to ApB$ (f) $ApB \to A, p$ (g) $ApB, p \to B$

3 Comparing DT proof systems and LK proof systems

LK is the usual Gentzen sequent calculus for Boolean formulas over the basis $\wedge$ and $\vee$. The *Boolean formulas* are defined inductively by

- Any literal p is a Boolean formula, and
- If A and B are Boolean formulas, then so are $(A \vee B)$ and $(A \wedge B)$.

The proof system LK has the same initial sequents (axioms) as LDT, its inference rules are the contraction rules *c-l* and *c-r*, the weakening rules *w-l* and *w-r*, the cut rule, and the following Boolean rules:

$$\wedge\text{-}l\text{:}\ \frac{A, B, \Gamma \to \Delta}{A \wedge B, \Gamma \to \Delta} \qquad \wedge\text{-}r\text{:}\ \frac{\Gamma \to \Delta, A \qquad \Gamma \to \Delta, B}{\Gamma \to \Delta, A \wedge B}$$

$$\vee\text{-}l\text{:}\ \frac{A, \Gamma \to \Delta \qquad B, \Gamma \to \Delta}{A \vee B, \Gamma \to \Delta} \qquad \vee\text{-}r\text{:}\ \frac{\Gamma \to \Delta, A, B}{\Gamma \to \Delta, A \vee B}$$

Recall that a *clause* is a disjunction of literals and a *term* is a conjunction of literals.

▸ **Definition 7.** *A Boolean formula is* depth one *if it is either a clause or a term.* 1-LK *is the fragment of* LK *in which all formulas appearing in sequents are depth one formulas.* Tree-1-LK *is the same system with the restriction that proofs are tree-like.*

If $\vec{p}$ is a vector of literals, we write $\bigvee \vec{p}$ to denote a disjunction of the literals $\vec{p}$, taken in the indicated order. The notation $\bigwedge \vec{p}$ is defined similarly. The nesting of disjunctions and conjunctions can be arbitrary, so $\bigvee \vec{p}$ denotes any formula of the form $(\bigvee \vec{p}\,') \vee (\bigvee \vec{p}\,'')$ where $\vec{p}\,'$ and $\vec{p}\,''$ denote $p_1, \ldots, p_k$ and $p_{k-1}, \ldots, p_\ell$ for some $1 \leqslant k \leqslant \ell$. Although these notations are ambiguous about the nesting of disjunctions or conjunctions, this makes no difference in this work, since if A and B are both of the form $\bigvee \vec{p}$ but with different orders of applications of $\vee$'s, then there are polynomial size, cut-free Tree-1-LK proofs of $A \to B$ and $B \to A$.

Later theorems will compare the proof theoretic strengths of various fragments and extensions of LDT to fragments of LK. Since these theories use different languages, we need to establish translations between cedents of DT formulas and (depth one) Boolean formulas.

▸ **Definition 8.** *For a (nonempty) sequence of literals $\vec{p}$ we define the DT formulas* $\mathrm{Conj}(\vec{p})$ *and* $\mathrm{Disj}(\vec{p})$ *by induction on the length of $\vec{p}$ as follows:*

$$\begin{array}{rclcrcl} \mathrm{Conj}(p) & := & p & & \mathrm{Disj}(p) & := & p \\ \mathrm{Conj}(p,\vec{p}) & := & (pp\mathrm{Conj}(\vec{p})) & & \mathrm{Disj}(p,\vec{p}) & := & (\mathrm{Disj}(\vec{p})pp) \end{array}$$

In other words, if $\vec{p} = (p_1, \ldots, p_\ell)$, for $\ell > 1$, we have:

$$\begin{aligned} \mathrm{Conj}(\vec{p}) &= (p_1p_1(p_2p_2(\cdots(p_{\ell-2}p_{\ell-2}(p_{\ell-1}p_{\ell-1}p_\ell))\cdots))) \\ \mathrm{Disj}(\vec{p}) &= (((\cdots((p_\ell p_{\ell-1}p_{\ell-1})p_{\ell-2}p_{\ell-2})\cdots)p_2p_2)p_1p_1). \end{aligned}$$

It is not hard to verify that Conj and Disj correctly express the conjunction and disjunction of the literals $\vec{p}$. This is borne out by the next proposition.

▸ **Proposition 9.** *The following sequents have polynomial size, cut-free* Tree-LDT *proofs.*

(a) $\mathrm{Conj}(\vec{p},\vec{q}) \to \mathrm{Conj}(\vec{p})$
(b) $\mathrm{Conj}(\vec{p},\vec{q}) \to \mathrm{Conj}(\vec{q})$
(c) $\mathrm{Conj}(\vec{p}), \mathrm{Conj}(\vec{q}) \to \mathrm{Conj}(\vec{p},\vec{q})$
(d) $\mathrm{Disj}(\vec{p}) \to \mathrm{Disj}(\vec{p},\vec{q})$
(e) $\mathrm{Disj}(\vec{q}) \to \mathrm{Disj}(\vec{p},\vec{q})$
(f) $\mathrm{Disj}(\vec{p},\vec{q}) \to \mathrm{Disj}(\vec{p}), \mathrm{Disj}(\vec{q})$

For the converse direction of simulating LDT (and its supersystems) by LK, we need to express DT formulas A as Boolean formulas in both CNF and DNF forms. For this we define $\mathrm{Tms}(A)$ as a multiset of terms (i.e., a multiset of conjunctions) and $\mathrm{Cls}(A)$ as a multiset of clauses (i.e., a multiset of disjunctions) so that A is equivalent to both the DNF $\bigvee \mathrm{Tms}(A)$ and the CNF $\bigwedge \mathrm{Cls}(A)$.

▸ **Definition 10.** *Let A be a* DT*-formula. The* terms *and* clauses *of A are the multisets* $\mathrm{Tms}(A)$ *and* $\mathrm{Cls}(A)$ *inductively defined by letting* $\mathrm{Tms}(p)$ *and* $\mathrm{Cls}(p)$ *both equal p, and letting*

$$\mathrm{Tms}(BpC) := \{(\overline{p} \wedge D) : D \in \mathrm{Tms}(B)\} \cup \{(p \wedge D) : D \in \mathrm{Tms}(C)\} \quad (2)$$

$$\mathrm{Cls}(BpC) := \{(p \vee D) : D \in \mathrm{Cls}(B)\} \cup \{(\overline{p} \vee D) : D \in \mathrm{Cls}(C)\}. \quad (3)$$

For example, if A is $p_1p_2(p_3p_4p_5)$ then $\mathrm{Tms}(A)$ is $\{\overline{p_2} \wedge p_1,\ p_2 \wedge \overline{p_4} \wedge p_3,\ p_2 \wedge p_4 \wedge p_5\}$, and $\mathrm{Cls}(A)$ is equal to $\{p_2 \vee p_1,\ \overline{p_2} \vee p_4 \vee p_3,\ \overline{p_2} \vee \overline{p_4} \vee p_5\}$.

The equivalence between A, $\bigvee \mathrm{Tms}(A)$ and $\bigwedge \mathrm{Cls}(A)$ is witnessed by simple proofs:

▸ **Proposition 11.** *There are polynomial size, cut-free* Tree-LK*-proofs of:*

(a) $C \to D$, *for each* $C \in \mathrm{Tms}(A)$ *and* $D \in \mathrm{Cls}(A)$.
(b) (i) $\mathrm{Cls}(ApB) \to D, p$, *for each* $D \in \mathrm{Cls}(A)$;
(ii) $p, \mathrm{Cls}(ApB) \to D$, *for each* $D \in \mathrm{Cls}(B)$.
(iii) $\mathrm{Cls}(A) \to D, p$, *for each* $D \in \mathrm{Cls}(ApB)$.
(iv) $p, \mathrm{Cls}(B) \to D$, *for each* $D \in \mathrm{Cls}(ApB)$.
(c) (i) $C \to p, \mathrm{Tms}(ApB)$, *for each* $C \in \mathrm{Tms}(A)$;
(ii) $p, C \to \mathrm{Tms}(ApB)$, *for each* $C \in \mathrm{Tms}(B)$.
(iii) $C \to p, \mathrm{Tms}(A)$, *for each* $C \in \mathrm{Tms}(ApB)$.
(iv) $p, C \to \mathrm{Tms}(B)$, *for each* $C \in \mathrm{Tms}(ApB)$.

Proof sketch. Part (a) of the lemma is proved by induction on the complexity of A. Parts (b) and (c) are trivial once the definitions are unwound. For example, (b.i) follows from the fact that $\mathrm{Cls}(ApB)$ contains the formula $p \vee D$. This allows (b.i) to be derived from the two sequents $p \to p$ and $D \to D$. The former is an axiom, and the latter has a tree-like cut-free proof by Proposition 6(a). The other cases are similar. ◂

The next definition shows how to compare proof complexity between proof systems that work with DT formulas and ones that work with Boolean formulas.

▸ **Definition 12.** *Let P be a proof system for sequents of Boolean formulas (or at least, sequents of depth one Boolean formulas), and Q be a proof system for sequents of DT formulas. We say that P* polynomially simulates *Q if there is a polynomial time procedure which, given a Q-proof of*

$$A_0, \ldots, A_{m-1} \to B_0, \ldots, B_{n-1}, \tag{4}$$

where the A_i's and B_i's are DT*-formulas, produces a P-proof of*

$$\mathrm{Cls}(A_0), \ldots, \mathrm{Cls}(A_{m-1}) \to \mathrm{Tms}(B_0), \ldots, \mathrm{Tms}(B_{n-1}). \tag{5}$$

The system Q polynomially simulates *P if there is a polynomial time procedure which, given a P-proof of*

$$\bigvee \vec{a}_0, \ldots, \bigvee \vec{a}_{m-1} \to \bigwedge \vec{b}_0, \ldots, \bigwedge \vec{b}_{n-1}, \tag{6}$$

where the $\vec{a}_i$'s and $\vec{b}_i$'s are sequences of literals, produces a Q-proof of

$$\mathrm{Disj}(\vec{a}_0), \ldots, \mathrm{Disj}(\vec{a}_{m-1}) \to \mathrm{Conj}(\vec{b}_0), \ldots, \mathrm{Conj}(\vec{b}_{n-1}). \tag{7}$$

The systems P and Q are polynomially equivalent *if they polynomially simulate each other. (5) is called the* Boolean translation *of (4). (7) is called the* DT-translation *of (6).* Quasipolynomial simulation and equivalence *are defined in the same way, but using quasipolynomial time (time $2^{\log^{O(1)} n}$) procedures.*[2]

3.1 1-LK and LDT

Our first results compare the weakest systems considered in this work, operating with just DT formulas or with just terms and clauses.

▸ **Theorem 13.** LDT *polynomially simulates* 1-LK*.* Tree-LDT *polynomially simulates* Tree-1-LK*.*

Proof sketch. We may replace terms $\bigwedge \vec{a}$ and clauses $\bigvee \vec{a}$ occurring in a 1-LK proof by DT-formulas $\mathrm{Conj}(\vec{a})$ or $\mathrm{Disj}(\vec{a})$ respectively. The result can be adapted into a correct LDT proof using cuts against proofs from Proposition 9. ◂

A converse result holds too, but we have only a quasipolynomial simulation in the tree-like case. It is open whether this can be improved to a polynomial simulation.

[2] It turns out that all stated quasipolynomial simulations in this work (Theorems 14 and 30) take time $n^{O(\log n)} = 2^{O(\log^2 n)}$.

▸ **Theorem 14.** 1-LK *polynomially simulates* LDT. Tree-1-LK *quasipolynomially simulates* Tree-LDT.

Proof sketch. In a given LDT proof, we may replace every DT A in an antecedent by the multiset $\mathrm{Cls}(A)$ and every DT A in a succedent by $\mathrm{Tms}(A)$. The result can be adapted into a correct 1-LK proof using cuts against proofs of the truth conditions from Proposition 11.

In the tree-like case, when simulating the cut rule we must copy one subproof polynomially many times (such copying is unnecessary when proofs are dag-like). However it turns out we may freely choose which of the two subproofs to duplicate, so we may just take the smaller one, which has size at most half that of the original proof. Doing this recursively yields a $n^{O(\log n)} = 2^{O(\log^2 n)}$ bound on the size of the resulting Tree-1-LK proof. ◂

4 Nondeterministic decision tree formulas and LNDT proofs

This section defines nondeterministic decision tree (NDT) formulas, and the associated sequent calculus LNDT. The NDT formulas have two kinds of connectives; the 3-ary case function ApB and the Boolean OR-gate ($\vee$). Formally:

▸ **Definition 15.** *The* nondeterministic decision tree formulas*, or* NDT formulas *for short, are inductively defined by*

- *Any literal p is a* NDT *formula;*
- *If A and B are* NDT *formulas and p is a literal, then (ApB) is a* NDT *formula;*
- *If A and B are* NDT *formulas, then $(A \vee B)$ is an* NDT *formula.*

A nondeterministic gate in a decision tree accepts just when at least one of its children is accepting. This corresponds exactly to an $\vee$ gate, which yields *True* exactly when at least one input is *True*. One of our motivations in defining LNDT that is will serve as a foundation for our later definition eLNDT, which will capture a logic for nondeterministic branching programs, and hence a logic for nonuniform NL.

▸ **Definition 16.** *The sequent calculus* LNDT *is a proof system in which lines are sequents of* NDT *formulas. Its initial sequents (axioms) and rules are the sames as those of* LDT*, along with the two $\vee$ inferences, $\vee$-l and $\vee$-r, of* LK *as described on page 6.*

For α a 0-1-truth assignment, the semantics of NDT formulas is defined extending the definition of the semantics of DT formulas, in equations 1, to include

$$\alpha(A \vee B) = \begin{cases} 1 & \text{if } \alpha(A) = 1 \text{ or } \alpha(B) = 1 \\ 0 & \text{otherwise.} \end{cases}$$

It is straightforward to verify that LNDT is sound and complete for sequents of NDT formulas, by a similar argument to that of Remark 5.

4.1 LDT and tree-like LNDT are equivalent

Next we turn to the relative complexity of LDT and LNDT. Naturally the latter subsumes the former, but this can be strengthened as follows:

▸ **Theorem 17.** Tree-LNDT *is polynomially equivalent to* LDT *over* DT*-sequents.*

We will soon see that this also refines the known polynomial equivalence between 1-LK and Tree-2-LK (see [2, 3]), by virtue of Theorems 14 and 18.

Proof sketch. To show that Tree-LNDT polynomially simulates LDT we notice that lines of an LDT proof (i.e. sequents of DT formulas) may be expressed as NDT formulas. From here one may use an adaptation of a standard technique for showing that tree-like LK is equivalent to dag-like LK, carefully managing the complexity of formulas occurring.

To show that LDT polynomially simulates Tree-LNDT, we first notice that each NDT formula may be written as a disjunction of DT formulas ('normal form'), and furthermore that LNDT proofs may be written in a way that operates with only such formulas with only polynomial blowup. Now we convert a normal form Tree-LNDT proof π of $\bigvee \Pi_1, \ldots, \bigvee \Pi_k \to \bigvee \Lambda_1, \ldots, \bigvee \Lambda_l$ to a (dag-like) LDT derivation π' of the sequent $\to \Lambda_1, \ldots, \Lambda_l$ from extra *hypotheses* $\{\to \Pi_i\}_{i=1}^k$. This is proved by induction on the structure of the proof tree and takes polynomial time. Now, when π derives a DT sequent, notice that π' is just a LDT proof of the same sequent. ◂

4.2 Equivalence of LNDT and 2-LK

A Boolean formula is *depth two* if it is depth one, or if it is a conjunction of clauses or a disjunction of terms. 2-LK is the fragment of LK in which all formulas occurring are depth two formulas. Tree-2-LK is the same system with the restriction that proofs are tree-like.

▸ **Theorem 18.** LNDT *and* 2-LK *are polynomially equivalent.* Tree-LNDT *and* Tree-2-LK *are polynomially equivalent.*

This is not so surprising a result, since NDTs have equivalent expressive power to DNFs, so depth two sequents may be written as NDT sequents and vice-versa.

Proof sketch. A (two-sided) 2-LK proof is simulated in LNDT by simply replacing every DNF $\bigvee_i \bigwedge \vec{p}_i$ with the NDT $\bigvee_i \mathrm{Conj}(\vec{p}_i)$ and locally repairing the proof using cuts against proofs from Proposition 8. In the other direction we work with 'normal form' LNDT proofs (as in the proof of Theorem 17). From here the translation to DNFs is straightforward, since DT formulas already have small DNFs, cf. Definition 10. Again, we use cuts against proofs of the appropriate truth conditions. Both simulations map tree-like proofs to tree-like proofs. ◂

5 Proof systems for branching programs

5.1 Formulas and proofs with extension variables

We now describe the systems eLDT and eLNDT which reason about deterministic and nondeterministic branching programs respectively.[3] Formulas can now include *extension variables*, usually denoted by e_1, e_2, etc. It is important that the extension variables are explicitly distinguished from the propositional variables we have thus far used.

The purpose of extension variables is to serve as abbreviations for more complex formulas. Thus, proofs that use extension variables will be accompanied by a set of extension axioms $\{e_i \leftrightarrow A_i\}_{i<n}$, where each formula A_i may use any literals p but is restricted to use only the extension variables e_j for $j < i$. The intent is that e_i is an abbreviation for the formula A_i.

▸ **Definition 19.** Extended decision tree *formulas (*eDT *formulas) are defined as follows:*

(1) *Any literal p is an* eDT *formula.*

[3] These systems could equally well be called LBP and LNBP, using "BP" for "branching programs'.

(2) *Any extension variable e is an* eDT *formula.*
(3) *If A and B are* eDT *formulas and p is a literal, then (ApB) is a* DT *formula.*

In particular, a decision literal p in a formula ApB is *not* allowed to be an extension variable. The intuition is that the extension variables may 'name' nodes in a branching program.

▸ **Definition 20.** Extended nondeterministic decision tree *formulas (*eNDT *formulas) are defined by the closure conditions (1)-(3) above (replacing "*eDT*" by "*eNDT*") and:*

(4) *If A and B are* eNDT *formulas, then $(A \vee B)$ is an* eNDT *formula.*

A set of *extension axioms* is a set $\mathcal{A} = \{e_i \leftrightarrow A_i\}_{i<n}$ where $e_0, \ldots, e_{n-1}$ are extension variables such that the only extension variables appearing in A_i are $e_0, \ldots, e_{i-1}$, for $i < n$. We identify $\mathcal{A}$ with the set of sequents consisting of $e_i \rightarrow A_i$ and $A_i \rightarrow e_i$, for $i < n$. eDT and eNDT formulas have truth semantics only relative to a set of extension axioms $\{e_i \leftrightarrow A_i\}_{i<n}$. Namely, for α a truth assignment, the definition of truth is extended by setting $\alpha(e_i) = \alpha(A_i)$.

▸ **Definition 21.** *An* eLDT *proof is a pair $(\pi, \mathcal{A})$ where $\mathcal{A} = \{e_i \leftrightarrow A_i\}_{i<n}$ is a set of extension axioms where each A_i is an* eDT *formula, and π is an* LDT *derivation which is allowed to use initial sequents from $\mathcal{A}$.* eLNDT *proofs are defined similarly, but with* eLNDT *formulas A_i and* eLNDT *derivations.*

Note that all formulas in an eLDT or eLNDT proof are based on a single set of extension axioms $\{e_i \leftrightarrow A_i\}_{i<n}$.

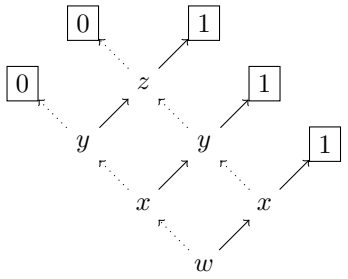
Let us discuss how the extended formulas we have introduced may be used to represent bona fide branching programs. A (deterministic) branching program is a directed acyclic graph G such that (a) G has a unique source node, (b) sink nodes in G are labelled with either 0 or 1, (c) all other nodes are labelled with a literal p and have two outgoing edges, one labelled 0 and the other 1. G can be converted into an equivalent eDT formula with associated extension axioms $\{e_i \leftrightarrow A_i\}_{i<n}$ by introducing an extension variable for every internal node in G. Conversely, as is described in more detail in Section 5.2, any eDT formula A with extension axioms $\{e_i \leftrightarrow A_i\}_{i<n}$ can be straightforwardly transformed into a linear size deterministic branching program. For this, the nodes in the branching program correspond to the extension variables e_i and the subformulas of the formulas A_i.

Nondeterministic branching programs are defined similarly to deterministic branching programs, but further allowing the internal nodes of G to be labelled with "$\vee$" as well as literals (in this case the labelling of its outgoing edges is omitted). The semantics is that an $\vee$-node is accepting provided at least one of its children is accepting. It is straightforward to convert a nondeterministic branching program into an eLNDT formula with associated extension axioms, and vice versa. A similar construction yields the folklore fact that 'extended Boolean formulas' are as expressive as Boolean circuits.

▸ **Example 22.** Consider the (deterministic) branching program G in Figure 2, on the left, which returns 1 just if at least two out of the four input variables w, x, y, z are 1. Edges labelled with 0 are here dotted (and always left outgoing) while edges labelled 1 are here solid (and always right outgoing). In this particular case, the branching program is *ordered* (or an *OBDD*), i.e. variables occur in the same relative order on each path from the source to a sink. The program also happens to compute a monotone Boolean function.

To represent G in eLDT, we introduce extension variables for each internal node of the program as follows. Write e_{ij} for the jth node of the ith layer, with i, j ranging from 0 onward, and introduce the extension axioms in Figure 2, on the right.[4] Now G is represented

[4] Formally, we are writing 0 and 1 as shorthand for $pp\bar{p}$ and $\bar{p}pp$ respectively, for some/any literal p.



$$
\begin{aligned}
e_{10} &\leftrightarrow e_{20} x e_{21} \\
e_{11} &\leftrightarrow e_{21} x 1 \\
e_{20} &\leftrightarrow 0 y e_{31} \\
e_{21} &\leftrightarrow e_{31} y 1 \\
e_{31} &\leftrightarrow 0 z 1
\end{aligned}
$$

Figure 2 A branching program G, on the left, computing the 2-out-of-4 threshold function and an encoding of its (internal) local conditions by extension variables, on the right. Dotted edges are labelled 0 and solid edges are labelled 1. G is equivalent to the eDT formula $e_{10} w e_{11}$.

by the eDT formula $e_{10} w e_{11}$. Notice that the orderedness of the program is reflected in its eLDT representation: writing (x_0, x_1, x_2, x_3) for (w, x, y, z), we have that x_i is the root of the formula that any e_{ij} abbreviates.

Other representations of G are possible, for instance by renaming the extension variables or by partially unwinding the graph. In both these two latter cases, the eDT representation obtained will be provably equivalent to the one above, by polynomial-size proofs in eLDT, by virtue of Lemma 28 later.

5.2 Foundational issues and Boolean combinations

The fact that extension variables cannot be used as decision literals is a significant limitation on the expressiveness of DT formulas. Recall for instance that the conjunction of p_1 and p_2 can be expressed with the DT formula $\mathrm{Conj}(p_1, p_2)$, namely $(p_1 p_1 p_2)$. However, it is not permitted to form $(e_1 e_1 e_2)$; in fact, it is not possible to express the conjunction $e_1 \wedge e_2$ without taking the extension axioms defining e_1 and e_2 into account. If we could write the conjunction of e_1 and e_2 by a generic formula $A(e_1, e_1)$, then we could introduce a new extension variable representing $A(e_1, e_2)$. This would imply that eDT formulas are as expressive as extended Boolean formulas; in other words, that deterministic branching programs would be as expressive as Boolean circuits. This is a non-uniform analogue of L = P (i.e., log-space equals polynomial time) and, of course, is an open question.

Nonetheless, for any given extension variables e and e', there is a formula $\textsc{And}(e, e')$ expressing the conjunction of e and e' by changing the underlying set of extension axioms. The intuition is that we start with the branching program G for e, but now with sink nodes labelled with 0 or 1 instead of with variables. To form the branching program for $e \wedge e'$, we take (an isomorphic copy) of the branching program G' for e', and modify G by replacing each sink node labelled with 1 with the source node of G' (in other words, each edge directed into a sink "1" is modified to instead point to the root of G'). Since we do not actually have 0 and 1 in the language, we work modulo their encodings by literals:

▸ **Definition 23.** *Let C be an* eDT *or* eNDT *formula. $C[0/B]$ is the formula obtained by replacing (in parallel) each occurrence of a literal p as a leaf in C with the formula $(B\,p\,p)$. Similarly, $C[1/B]$ is the formula obtained by replacing each occurrence of a literal p as a leaf in C with the formula $(p\,p\,B)$.*

The point of $C[0/B]$ is that $(B\,p\,p)$ evaluates to 1 if p is true, and to B otherwise. Thus, the intent is that $C[0/B]$ is equivalent $C \vee B$. Likewise, we want $C[1/B]$ to be equivalent

$C \wedge B$. However, these equivalences hold only if the substitutions are applied not just in C but instead throughout the definitions of the extension axioms used in C. This is done with the following definition.

▸ **Definition 24.** *Let $\mathcal{A}$ be a set of extension axioms $\{e_i \leftrightarrow A_i\}_{i<n}$. Another set of extension axioms $\mathcal{A}[1/B]$ is defined as follows. First, let $\{e'_i\}_i$ be a set of* new *extension variables. Define $A_i[\vec{e}'/\vec{e}]$ to be the result of replacing each e_j in A_i with e'_j. Let A'_i be $(A_i[\vec{e}'/\vec{e}])[1/B]$. Then $\mathcal{A}[1/B]$ is the set of extension axioms $\{e'_i \leftrightarrow A'_i\}_{i<n} \cup \mathcal{A}$. The set $\mathcal{A}[0/B]$ is defined similarly: letting $\vec{e}''$ be another set of new extension variables, defining A''_i to be $(A_i[\vec{e}''/\vec{e}])[0/B]$, and letting $\mathcal{A}[0/B]$ be the set of extension axioms $\{e''_i \leftrightarrow A''_i\}_{i<n} \cup \mathcal{A}$.*

Finally, if A and B are eDT *or* eNDT *formulas defined using extension axioms $\mathcal{A}$, then $\textsc{And}(A, B)$ is by definition $A[1/B]$ relative to the extension axioms $\mathcal{A}[1/B]$. The formula $\textsc{Or}(A, B)$ for disjunction is defined similarly, namely, it is equal to $A[0/B]$ relative to the extension axioms $\mathcal{A}[0/B]$.*

Note the two formulas $\textsc{And}(A, B)$ and $\textsc{Or}(A, B)$ introduced *different* sets of new extension variables, so we may use both $\textsc{And}(A, B)$ and $\textsc{Or}(A, B)$ without any clashes between extension variables. More generally, we adopt the convention that the new extension variables are uniquely determined by the Boolean combination being constructed. For instance, e'_i could have instead been designated $e_{i,(A\wedge B)}$. When measuring proof size, we also need to count the sizes of the subscripts on the extension variables. This clearly however only increases proof size polynomially.

There are two other sources of growth of size in forming $\textsc{And}(A, B)$ and $\textsc{Or}(A, B)$. The first is that formula sizes increase since copies of B is substituted in at many places in A and $\mathcal{A}$: this potentially gives a quadratic blowup in proof size. We avoid this quadratic blowup in proof size, by always taking B to be a single variable (namely, an extension variable). The construction of $\textsc{And}(A, B)$ or $\textsc{Or}(A, B)$ also introduces many new extension variables, namely it potentially doubles the number of variables. To control this, we will ensure that the constructions of $\textsc{And}(\cdot, \cdot)$ and $\textsc{Or}(\cdot, \cdot)$ are nested only logarithmically.

▸ **Example 25.** Consider the formula $\textsc{And}(p_1, \textsc{And}(p_2, p_3))$, which is a translation of the Boolean formula $p_1 \wedge (p_2 \wedge p_3)$ to a DT formula. To form $\textsc{And}(p_2, p_3)$, start with $(p_2 p_2 1)$ and substitute p_3 for "1", to obtain $(p_2 p_2 p_3)$. Then $\textsc{And}(p_1, \textsc{And}(p_2, p_3))$ is obtained by forming $(p_1 p_1 1)$ and replacing "1" with $\textsc{And}(p_2, p_3)$ to obtain $(p_1 p_1 (p_2 p_2 p_3)$. It is also the same as $\mathrm{Conj}(p_1, p_2, p_3)$. A similar construction shows that $\textsc{Or}(p_1, \textsc{Or}(p_2, p_3)$ is equal to $((p_3 p_2 p_2) p_1 p_1)$. This is a translation of the Boolean formula $p_1 \vee (p_2 \vee p_3)$ to a DT formula, and is equal to $\mathrm{Disj}(p_1, p_2, p_3)$.

▸ **Example 26.** Let A be the formula $(p_1 p_2 (e_1 p_3 e_2))$ and B be the formula $(q_1 q_2 e_2)$ in the context of the extension axioms $\mathcal{A}$

$$e_1 \leftrightarrow (r_1 \overline{r_2} e_2) \qquad e_2 \leftrightarrow (\overline{s_1} s_2 s_3), \tag{8}$$

where p_i, q_i, r_i, s_i are literals. The formula $A[0/B]$ is formed as follows. First $\mathcal{A}(\vec{e}'/\vec{e})$ is $e'_1 \leftrightarrow (r_1 \overline{r_2} e'_2)$, $e'_2 \leftrightarrow (\overline{s_1} s_2 s_3)$. Then $\mathcal{A}[0/B]$ contains the extension axioms of $\mathcal{A}$ as shown in (8) plus the extension axioms $e'_1 \leftrightarrow ((B r_1 r_1) \overline{r_2} e'_2)$, $e'_2 \leftrightarrow ((B\,\overline{s_1}\,\overline{s_1}) s_2 (B s_3 s_3))$. Finally, $A[0/B]$ is the DT formula $((B p_1 p_1) p_2 (e'_1 p e'_2))$, namely, $(((q_1 q_2 e_2) p_1 p_1) p_2 (e'_1 p e'_2))$, relative to the four extension axioms in $\mathcal{A}[0/B]$.

5.3 Truth conditions and renaming of extension variables

We show that, despite the delicate renaming of variables required for notions such as $A[0/B]$ and $\textsc{And}(A, B)$, for DT (respectively eNDT) formulas A, B, we may nonetheless realise their

basic truth conditions by small eLDT (respectively eLNDT) proofs:

▸ **Lemma 27.** *Let A and B be* eDT *formulas (respectively,* eNDT *formulas) relative to extensions axioms $\mathcal{A}$. Then, the sequents* (a)-(c) *below have polynomial size, cut free* eLDT *proofs (respectively,* eLNDT *proofs) relative to the extension axioms $\mathcal{A}[0/B]$. The same holds for the sequents* (d)-(f) *relative to $\mathcal{A}[1/B]$.*

(a) $B \to A[0/B]$ (b) $A \to A[0/B]$ (c) $A[0/B] \to A, B$ (d) $A[1/B] \to B$ (e) $A[1/B] \to A$ (f) $A, B \to A[1/B]$

Proof sketch. Parts (a)-(c) are proved by showing inductively that if C is a subformula of $A[0/B]$ or a subformula of any A'_i in $\mathcal{A}[0/B]$, then $C \to A, B$ and $B \to C$ and $A \to C$ have short eLDT (resp., eLNDT) proofs. The base cases are just the cases where C is is the form $(B\,p\,p)$. The inductive cases are trivial. A similar argument proves cases (d)-(f). ◂

The proofs of Lemma 27 seem to be inherently dag-like, and we do not know if there are polynomial-size Tree-eLDT proofs for those sequents.

As discussed above, we assume that the choice of new extension variables $\vec{e}'$ or $\vec{e}''$ depends explicitly on what formula $\textsc{And}(A, B)$ and $\textsc{Or}(A, B)$ is being formed. In other words, each e'_i or e''_i is a variable $e_{i,\textsc{And}(A,B)}$ or $e_{i,\textsc{Or}(A,B)}$. In the proof of Theorem 29 later, this means that the translations of distinct occurrences of the same Boolean formula use the same extension variables. However, this is not strictly necessary, as eLDT can prove the equivalence of formulas after renaming extension variables:

▸ **Lemma 28.** *Suppose A is a* DT *formula w.r.t. extension axioms $\mathcal{A} = \{e_i \leftrightarrow A_i\}_i$, and that the extension variables $\vec{f}$ are distinct from the extension variables $\vec{e}$. Let B equal $A[\vec{f}/\vec{e}]$ w.r.t. the extension axioms $\mathcal{B} = \{f_i \leftrightarrow A_i[\vec{f}/\vec{e}]\}_i$. Then* eLDT *has a polynomial size, cut free (dag-like) proofs of $A \to B$ and $B \to A$ relative to the extension axioms $\mathcal{A} \cup \mathcal{B}$.*

Lemma 28 has a straightforward proof that proceeds inductively through all subformulas of the formulas A_i and A.

6 Simulations for eLDT, eLNDT and LK

We compare the systems eLDT and and eLNDT with LK, showing that they are all quasi-polynomially related in terms of proof size, constituting the upper half of Figure 1.

6.1 eLDT polynomially simulates LK

The intuition for the next simulation is that the formulas in an LK proof are Boolean and may be evaluated in log-space. Thus they may be expressed by polynomial-size eDT formulas (under appropriate extension axioms).

▸ **Theorem 29.** eLDT *(and so also* eLNDT*) polynomially simulates* LK*.*

Proof sketch. We assume the given LK proof is written in *balanced* form, i.e. with only $O(\log n)$-depth Boolean formulas occurring. Once again we proceed by replacing each formula occurrence by an eDT formula representing it, by virtue of the constructions of And and Or from Definition 24. (We appeal to the logarithmic depth of Boolean formula occurrences in order to control the complexity of this translation). From here we locally simulate each step of the LK proof by cutting against the truth conditions from Lemma 27. ◂

6.2 LK quasipolynomially simulates eLNDT

The intuition for the next simulation is that eNDT formulas define nondeterministic logspace properties, and these are expressible with quasipolynomial size Boolean formulas.

▸ **Theorem 30.** LK *quasipolynomially simulates* eLNDT *(and so also* eLDT*).*

Proof sketch. We work from the observation that NL predicates have quasipolynomial-size (in fact $n^{O(\log n)}$-size) Boolean formulas. Moreover, there is an *evaluator* for non-deterministic branching programs with quasipolynomial-size Boolean formulas for *st*-connectivity in graphs, whose basic properties were shown to have quasipolynomial-size LK proofs in [4]. Once the basic truth conditions of this evaluator are given appropriate LK proofs, we may proceed by duly replacing every eNDT formula occurrence in an eLNDT proof π by the corresponding Boolean formula evaluating the non-deterministic branching program it represents. We cut against proofs of the truth conditions to locally simulate each step of π. ◂

7 Conclusions

We presented sequent-style systems LDT, LNDT, eLDT and eLNDT that manipulate decision trees, nondeterministic decision trees, branching programs (via extension) and nondeterministic Branching Programs (via extension) respectively. The systems eLDT and eLNDT serve as natural systems for log-space and nondeterministic log-space reasoning, respectively. We examined their relative proof complexity and also compared them to (low depth) Frege systems (more precisely their representations in the sequent calculus LK).

We did not compare the proof complexity theoretic strength of our systems eLDT and eLNDT with the systems GL* for L and GNL* for NL in [31, 32]. In future work we intend to show that our systems correspond to the bounded arithmetic theories VL and VNL in the usual way. Namely, proofs of Π_1 formulas in VL translate to families of small eLDT proofs of each instance, and, conversely, VL proves the soundness of eLDT. (Similarly for VNL and eLNDT.) This would render our systems polynomially equivalent to GL* and GNL*, respectively, by the analogous results from [31, 32], though this remains work in progress.

Two natural open questions arise from this work.

▻ Question 31. Does Tree-1-LK polynomially simulate Tree-LDT, or is there a quasipolynomial separation between the two?

▻ Question 32. Does Tree-eLDT polynomially simulate eLDT? Similarly for eLNDT.

While well-defined, the systems Tree-eLDT and Tree-eLNDT do not seem very robust, in the sense that it is not immediate how to witness branching program isomorphisms with short proofs. Nonetheless, it would be good to settle their proof complexity theoretic status.

There has been much recent work on the proof complexity of systems that may manipulate OBDDs [24, 6, 20], branching programs where propositional variables must occur in the same relative order on each path through the dag. In fact, we could also define an 'OBDD fragment' of eLDT by restricting lines to eDT formulas expressing OBDDs, as alluded to in Example 22. It would be interesting to examine such systems from the point of view of proof complexity in the future, in particular comparing them to existing OBDD systems.

References

1 Toshiyasu Arai. A bounded arithmetic AID for Frege systems. *Annals of Pure and Applied Logic*, 103:155–199, 2000.

2 Arnold Beckmann and Samuel R. Buss. Separation results for the size of constant-depth propositional proofs. *Annals of Pure and Applied Logic*, 136:30–55, 2005.

3 Arnold Beckmann and Samuel R. Buss. On transformations of constant depth propositional proofs. *Annals of Pure and Applied Logic*, ??:???–???, 2019. To appear. doi:10.1016/j.apal.2019.05.002.

4 Sam Buss. Quasipolynomial size proofs of the propositional pigeonhole principle. *Theoretical Computer Science*, 576(C):77–84, 2015. doi:10.1016/j.tcs.2015.02.005.

5 Sam Buss, Anupam Das, and Alexander Knop. Proof complexity of systems of (non-deterministic) decision trees and branching programs, 2019. arXiv:1910.08503.

6 Sam Buss, Dmitry Itsykson, Alexander Knop, and Dmitry Sokolov. Reordering rule makes OBDD proof systems stronger. In *33rd Computational Complexity Conference, CCC 2018, June 22-24, 2018, San Diego, CA, USA*, pages 16:1–16:24, 2018. URL: https://doi.org/10.4230/LIPIcs.CCC.2018.16, doi:10.4230/LIPIcs.CCC.2018.16.

7 Samuel R. Buss. *Bounded Arithmetic*. Bibliopolis, Naples, Italy, 1986. Revision of 1985 Princeton University Ph.D. thesis.

8 Samuel R. Buss. The Boolean formula value problem is in ALOGTIME. In *Proceedings of the 19-th Annual ACM Symposium on Theory of Computing*, pages 123–131, May 1987.

9 Samuel R. Buss and Pavel Pudlák. How to lie without being (easily) convicted and the lengths of proofs in propositional calculus. In L. Pacholski and J. Tiuryn, editors, *Proceedings of the 8th Workshop on Computer Science Logic, Kazimierz, Poland, September 1994*, Lecture Notes in Computer Science #933, pages 151–162, Berlin, 1995. Springer-Verlag.

10 Stephen A. Cook. A survey of complexity classes and their associated propositional proof systems and theories, and a proof system for log space. Talk presented at the ICMS Workshop on Circuit and Proof Complexity, Edinburgh, October 2001. http://www.cs.toronto.edu/ sacook/.

11 Stephen A. Cook. Feasibly constructive proofs and the propositional calculus. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, pages 83–97. Association for Computing Machinery, 1975.

12 Stephen A. Cook and Antonina Kolokolova. A second-order system for polytime reasoning based on Grädel's theorem. *Annals of Pure and Applied Logic*, 124:193–231, 2003.

13 Stephen A. Cook and Antonina Kolokolova. A second-order theory for NL. In *Proc. 19th IEEE Symp. on Logic in Computer Science (LICS'04)*, pages 398–407, 2004.

14 Stephen A. Cook and Tsuyoshi Morioka. Quantified propositional calculus and a second-order theory for NC^1. *Archive for Mathematical Logic*, 44:711–749, 2005.

15 Stephen A. Cook and Phuong Nguyen. *Foundations of Proof Complexity: Bounded Arithmetic and Propositional Translations*. ASL and Cambridge University Press, 2010. 496 pages.

16 Stephen A. Cook and Robert A. Reckhow. On the lengths of proofs in the propositional calculus, preliminary version. In *Proceedings of the Sixth Annual ACM Symposium on the Theory of Computing*, pages 135–148, 1974.

17 Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44:36–50, 1979.

18 Martin Dowd. Propositional representation of arithmetic proofs. In *Proceedings of the 10th ACM Symposium on Theory of Computing (STOC)*, pages 246–252, 1978. doi:10.1145/800133.804354.

19 Erich Grädel. Capturing complexity classes by fragments of second order logic. *Theoretical Computer Science*, 101:35–57, 1992.

20 Dmitry Itsykson, Alexander Knop, Andrei E. Romashchenko, and Dmitry Sokolov. On obdd-based algorithms and proof systems that dynamically change order of variables. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany*, pages 43:1–43:14, 2017. URL: https://doi.org/10.4230/LIPIcs.STACS.2017.43, doi:10.4230/LIPIcs.STACS.2017.43.

21 Emil Jeřábek. Dual weak pigeonhole principle, Boolean complexity, and derandomization. *Annals of Pure and Applied Logic*, 124:1–37, 2004. doi:10.1016/j.apal.2003.12.003.

22 Jan Johannsen. Satisfiability problem complete for deterministic logarithmic space. In *Proc. 21st Symp. on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science 2996, pages 317–325. Springer, 2004.

23 Stasys Jukna, Alexander A. Razborov, Petr Savický, and Ingo Wegener. On P versus NP ∩ co-NP for decision trees and read-once branching programs. *Computational Complexity*, 8(4):357–370, 1999. URL: `https://doi.org/10.1007/s000370050005`, `doi:10.1007/s000370050005`.

24 Alexander Knop. IPS-like proof systems based on binary decision diagrams. Typeset manuscript, June 2017.

25 Jan Krajíček. *Bounded Arithmetic, Propositional Calculus and Complexity Theory*. Cambridge University Press, Heidelberg, 1995.

26 Jan Krajíček. *Proof Complexity*. Cambridge University Press, 2019.

27 Jan Krajíček and Pavel Pudlák. Quantified propositional calculi and fragments of bounded arithmetic. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 36:29–46, 1990.

28 Jan Krajíček and Gaisi Takeuti. On induction-free provability. *Annals of Mathematics and Artificial Intelligence*, pages 107–126, 1992.

29 Richard E. Ladner. The circuit value problem is log space complete for P. *SIGACT News*, 7:18–20, 1975.

30 Jeff B. Paris and Alex J. Wilkie. Counting problems in bounded arithmetic. In *Methods in Mathematical Logic, Lecture Notes in Mathematics #1130*, pages 317–340. Springer-Verlag, 1985.

31 Steven Perron. A propositional proof system for log space. In *Proc. 14th Annual Conf. Computer Science Logic (CSL)*, Springer Verlag Lecture Notes in Computer Science 3634, pages 509–524, 2005.

32 Steven Perron. *Power of Non-Uniformity in Proof Complexity*. PhD thesis, Department of Computer Science, University of Toronto, 2009.

33 G. S. Tsejtin. On the complexity of derivation in propositional logic. *Studies in Constructive Mathematics and Mathematical Logic*, 2:115–125, 1968.

34 Ingo Wegener. *Branching Programs and Binary Decision Diagrams*. SIAM, 2000. URL: `http://ls2-www.cs.uni-dortmund.de/monographs/bdd/`.