On Proof Systems Behind Efficient SAT Solvers

DoRon B. Motter and Igor L. Markov
University of Michigan, EECS
1301 Beal Ave
Ann Arbor, MI 48109-2122
{dmotter,imarkov}@eecs.umich.edu

Abstract

Conventional algorithms for Boolean satisfiability (SAT) work within the framework of resolution as a proof system. However it has been known since the 1980s that every resolution proof of the pigeonhole principle (PHP_n^m), suitably encoded as a CNF instance, includes exponentially many steps[4]. Therefore SAT solvers based upon the DLL procedure [1] or the DP procedure [2] must take exponential time. Polynomial-sized proofs of the PHP exist for more powerful proof systems, but general-purpose SAT solvers often remain confined to resolution.

Our work seeks automatizable proof systems other than resolution. In an earlier work [9], an implementation of a Compressed-BFS algorithm *empirically* solved $\overline{PHP_n^{n+1}}$ instances in $\Theta(n^4)$ time. Here, we add to this claim, and show *analytically* that these instances are solvable in polynomial time by Compressed-BFS. Thus the class of tautolgies efficiently provable by Compressed-BFS is different than that of resolution.

We hope that the details of our proof shed some light on the proof system implied by Compressed-BFS. Our proof focuses on structural invariants within the compressed data structure that stores collections of sets of open clauses during the Compressed-BFS algorithm. We bound the size of this data structure, as well as the overall memory, by a polynomial. Because of the nature of ZDD operations, the overall runtime is bounded by a polynomial of the data structure's size.

1 Introduction

State-of-the-art complete SAT solvers are based on the Davis-Logemann-Loveland (DLL) search procedure [1]. DLL is a backtracking algorithm several extensions, but its runtime on unsatisfiable instances is lower-bounded by the length of resolution proofs. This fact can be combined with known exponential lower bounds for resolution proofs of certain families of SAT instances, such as the pigeonhole instances. The result is that any implementation of the DLL algorithm must require exponential time on pigeonhole instances. Indeed, the well-known solvers Chaff [8] and GRASP [5] empirically take exponential time to solve these instances.

While much recent work was concerned with incremental improvements in implementation details of the DLL procedure, a different avenue of research is to look for new solver algorithms which lead to other classes of tractable problems. Put differently, we are looking for proof systems other than resolution which are simple enough to have reasonably efficient SAT solvers associated with them. To this end, we point out that the recently reported Compressed-BFS algorithm [9] empirically solves pigeonhole instances in polynomial time. Since Compressed-BFS is a general-purpose algorithm and competitively solves a number of standard benchmarks, two questions arise: (i) whether the empirical polynomial-time result for pigeonholes can be supported analytically, and (ii) what proof system is implied by Compressed-BFS.

To this end, our work offers the first analytical proof that the Compressed-BFS search procedure reported in [9] solves SAT instances from the pigeonhole family in polynomial time. This has immediate implications to the proof system behind the Compressed-BFS procedure. Since no polynomial-sized resolution proofs exist for the pigeonhole instances, then in this regard, the proof system must be more powerful than resolution. While we do not claim resolution is subsumed by Compressed-BFS, there is an infinite family of instances for which Compressed-BFS exponentially outperforms resolution.

Results such as this were known for other methods of solving SAT, such as *OBDD-apply*, which recursively constructs an OBDD of the given formula to determine satisfiability. It is known that this method and resolution cannot polynomially simulate each other [3]. Here we show that resolution cannot polynomially simulate Compressed-BFS, as it solves in polynomial time a class of formulas which require exponential length resolution proofs.

Unfortunately, explicitly formulating the unknown proof system appears difficult due to the complexity of Compressed-BFS procedure. However, we provide the details of our polynomiality proof for pigeonholes in the hope that they will shed some light on the unknown proof system or at least some of its features.

The remaining part of this paper is organized as follows. Section 2 reviews the necessary background, Section 3 introduces the classic pigeonhole instances and some of their relevant properties. The Compressed-BFS algorithm is described in Section 4. In Section 5 we show that this algorithm proves the pigeonhole principle in polynomial time. We give both the general outline of the proof and fairly detailed arguments for each step. Conclusions and our ongoing research are described in Section 6.

2 Background

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of Boolean variables. A *truth assignment* for V is a mapping $t : V \to \{0, 1\}$. A *partial truth assignment* for V is a truth assigment to some subset of variables $V' \subseteq V$.

A *clause* can be viewed a set of literals. A clause is *satisfied* by a truth assignment t iff at least one of its literals is true under t. A clause is said to be *violated* by a truth assignment t if all of its literals are false under t. A Boolean formula in conjunctive normal form can be represented by a set C of clauses.

The implicit representation used in the Compressed-BFS algorithm is dependent on the correspondence between valid partial truth assignments and sets of clauses. A partial truth assignment is said to be *invalid* if this assignment violates some clause. Given a valid partial truth assignment t, we can classify clauses in a CNF with respect to t as follows.

- If a clause has at least one literal assigned some value, and no literals are assigned true, this clause is said to be *open*.
- If any literals within a clause are assigned true, this clause is said to be *satisfied*.
- If no literals within a clause are assigned, the clause is said to be *not activated*.
- If all but one literal in a clause is assigned, the clause is said to be *unit*.

The compression in Compressed-BFS comes by storing the collection of sets of open clauses within a Zero Supressed Binary Decision Diagram (ZDD). ZDDs can be used to represent combinatorial objects by a set of paths in a Directed Acyclic Graph (DAG) [7, 6]. Since the number of paths in a DAG can be exponentially larger than the number of vertices, ZDDs are able to achieve an exponential level of compression in certain instances. Specifically, when the collection of sets is sparse or structured then the ZDD is often able to represent the entire collection compactly, and it is this compact representation which allows Compressed-BFS to refute $\overline{PHP_n^{n+1}}$ instances in time polynomial in n.

A ZDD is defined as a directed acyclic graph (DAG) where each node has a unique label, an integer index, and two outgoing edges which connect to what we will call *T-Child* and *E-child*. Because of this we can represent each node X as a 3-tuple $X\langle n, X_T, X_E\rangle$ where n is the index of the node X, X_T is its *T-Child*, and X_E is its *E-Child*. Each path in the DAG ends in one of two special nodes, the **0** node and the **1** node. These nodes have no successors. In addition, there is a single root node. When we use a ZDD we will in reality keep a reference to the root node. The semantics of a ZDD can be defined recursively by defining the semantics of a given node.

A ZDD can be used to encode a collection of sets by encoding its characteristic function. We can evaluate a function represented by a ZDD by traversing the DAG beginning at the root node. At each node X, if the variable corresponding to the index of X is true, we select the *T-Child*. Otherwise we select with the *E-Child*. Eventually we will reach either **0** or **1**, indicating the value of the function on this input. We augment this with the *Zero-Suppression Rule*: we may eliminate nodes whose $T - Child$ is **0**. With these standard rules, **0** represents the empty collection of sets, while **1** represents the collection consisting of only the empty set. ZDDs interpreted this way have a standard set of operations based on recursive definitions [6], including the union and intersection of two collections of sets, for example.

3 The CNF Instances $\overline{PHP_n^{n+1}}$

In this work we consider a CNF encoding of the negation of the pigeonhole principle. Such instances are easy to generate, widely available, and a part of standard SAT benchmark suites, where they are known as `hole-n` instances. The pigeonhole principle (PHP_n^m) states that if m pigeons ($m > n$) are placed in n holes, then some hole must contain more than one pigeon. Since PHP_n^{n+1} is valid, we can encode its negation, $\overline{PHP_n^{n+1}}$ in CNF form to obtain an unsatisfiable SAT instance. One way to do this is to use $n(n+1)$ variables, each representing that pigeon $j \le n+1$ is in hole $i \le n$. We can thus form n groups H_i of $n+1$ variables each, where each variable within H_i represents that some pigeon is in hole i. We can number variables in such a way that the first $n+1$ variables $x_1, x_2, \ldots, x_{n+1}$ make up the group H_0, and so on. Then, the group H_i is made up of the variables

$$H_i = \{x_{i(n+1)+1}, x_{i(n+1)+2}, \ldots, x_{(i+1)(n+1)}\}$$

Encoded this way, the constraints fall into two categories. Within each group of variables H_i, there are *pairwise exclusion* clauses of the form

$$(\bar{x}_{i(n+1)+a} + \bar{x}_{i(n+1)+b})$$

whenever $1 \le a \le n+1, 1 \le b \le n+1$ and $a \ne b$. These clauses prevent more than one pigeon from being in hole i since whenever any one variable $x_{i(n+1)+a} \in H_i$ is set true, there are *pairwise exclusion* clauses between $x_{i(n+1)+a}$ and every other variable in H_i. For these clauses to be satisfied then, at most one variable in H_i may be true or all such variables are set false.

The second category of constraints, *pigeon* clauses, are $n+1$ clauses of the form

$$(x_j + x_{(n+1)+j} + x_{2(n+1)+j} + \ldots + x_{(n-1)(n+1)+j})$$

which select the jth element from each H_i. These encode that each pigeon j must be in at least one hole: at least one of $x_j, x_{(n+1)+j}, \ldots, x_{(n-1)(n+1)+j}$ must be true for this clause to be satisfied.

To make this formulation clear, a schematic representation of the instance $\overline{PHP_2^3}$ is shown in Figure 1. Here, all variables are represented by dots on a horizontal line. The *pigeon* clauses are shown below this line, while the *pairwise exclusion* clauses are shown above this line. In Compressed-BFS, these variables are processed left-to-right, or equivalently, in increasing numerical order.

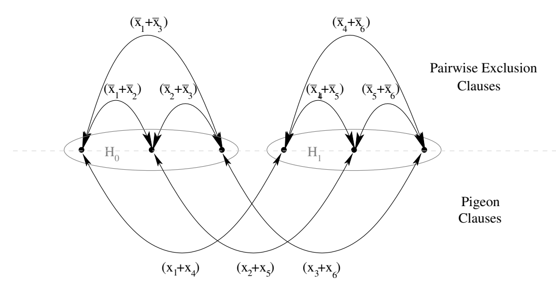

Figure 1: The Instance $\overline{PHP_2^3}$

3.1 Structure of the Instances $\overline{PHP_n^{n+1}}$

Some insight can be gained by a careful examination of the instances $\overline{PHP_n^{n+1}}$ which will be useful in the proof that Compressed-BFS provides a polynomial time refutation of these instances. It will also clarify the ideas which come into play at later stages of the proof.

We first introduce the notion of *the cut*. Since in Compressed-BFS we process variables according to some fixed order, it will happen that only certain clauses will be activated. However, if all variables in a clause are assigned, then for any branch in our search, this clause must be satisfied already. Clauses which have some, but not all variables assigned have the potential to affect our search, and are referred to as *cut* clauses. The term *cut* can be understood by drawing a vertical line on figure 1. Then when a partial truth assignment to variables to the left of our line is being considered, only clauses this line crosses (or *cuts*) will have the potential to be open clauses.

The first realization which will be useful was mentioned above: the *pairwise exclusion* clauses for a given set of variables H_i prevent more than one of variables from H_i from being assigned true. Thus, when performing a search for satisfiability, ideally there will be at most $n+2$ different 'branches' in the search: one branch corresponding to setting each of the $n+1$ variables in H_i true, and one final branch corresponding to setting them all false.

The second realization is that between H_i and H_{i+1}, the structure of $\overline{PHP_n^{n+1}}$ is considerably simpler. Figure 2 shows a schematic representation of a larger instance of $\overline{PHP_n^{n+1}}$ to highlight the general form. As seen in Figure 2, only the $n+1$ *pigeon* clauses extend between H_i and H_{i+1}. Immediately after variables in H_i, we say that only these $n+1$ *pigeon* clauses are in the *cut*. Since we will be considering the effects of Compressed-BFS over these instances, we will consider truth assignments to variables up to a given depth. If we look at the effects of truth assignments to all

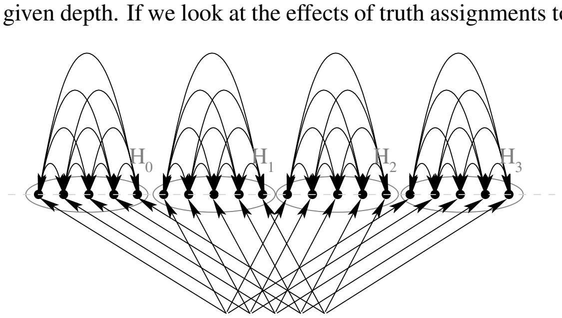

Figure 2: A Larger Instance: $\overline{PHP_4^5}$

variables in $H_1, H_2, \ldots, H_i$, then whatever effects these assignments have on the formula must be completely captured in these $n+1$ *pigeon* clauses. In Compressed-BFS, the number of clauses in the *cut* affects the performance of the algorithm [9]. We can consider all possible truth assignments to variables in $H_1, H_2, \ldots, H_i$ to deduce the following lemma.

Lemma 1. Let $k \in \{1, 2, \ldots, n\}$. After assigning values to variables $x_1, x_2, \ldots, x_{k(n+1)}$, we may satisfy at most k of the $n+1$ *pigeon* clauses.

Proof. First, notice that for $i < k$ at most one of variables in the set $H_i = \{x_{i(n+1)+1}, x_{i(n+1)+1}, \ldots, x_{(i+1)(n+1)}\}$ can be assigned *true*. This is because if two variables from H_i were true, then one of the *pairwise exclusion* clauses for H_i would be violated. As a result, any valid partial truth assignment to the first $k(n+1)$ variables must set only one variable in H_i true, for each i. By similarly examining clauses, it is evident that any partial truth assignment setting at most one of variables H_i true is valid. Setting all such variables to false does not violate any of the *pigeon* clauses, since we assume $k \leq n$. If exactly one of the variables within H_i is assigned true, then this simply satisfies one of the $n+1$ *pigeon clauses*, and maintains the validity of the partial truth assignment.

Since there are k sets of the form $H_i = \{x_{i(n+1)+1}, x_{i(n+1)+1}, \ldots, x_{(i+1)(n+1)}\}$, and each can satisfy at most one of the $n+1$ *pigeon* clauses, it is clear that we may satisfy at most k of these clauses.

4 The Compressed-BFS Algorithm Cassatt

The idea behind the Compressed-BFS algorithm *Cassatt* [9] corresponds directly to a BFS. Cassatt processes variables according to a static order, and implicitly represents all promising truth assignments of a given depth d. These *valid partial truth assignments* are assignments to variables $x_1, x_2, \ldots, x_d$ which do not cause all literals in some clause to be assigned false. The collection of these *partial truth assignments* is called the *front*. To determine the proper state after processing variable x_{d+1}, the algorithm 'copies' the *front*, and modifies one copy of each assignment within this collection to reflect the additional assignment of $x_{d+1} = true$. It modifies the other copy of the *front* to reflect assigning $x_{d+1} = false$. Finally, all *valid partial truth assignments* arising from either of these branches might yield satisfiability, so both branches are combined into the single new *front*.

Storing subsets of *open* clauses instead of explicit partial truth assignments is enough information to perform a BFS and determine satisfiability of a formula. Cassatt uses standard ZDD operations to maintain the collection of sets of open clauses, called the *front*. By combining this collection of sets of open clauses with a new truth assignment to a single variable, the *front* can be advanced as described above. To update the *front* to reflect a truth assignment to a single variable, the effects of this truth assignment on the status of clauses must be considered. In general, an assignment to a single variable $x_i = t$ (where $t \in \{0, 1\}$) has the following effects on clauses.

- It *violates* some clauses. Let $U_{x_i,t}$ be the set of *unit* clauses for which this variable assignment causes a conflict. Then, any subset in the *front* containing some $u \in U_{x_i,t}$ must be pruned as it cannot yield satisfiability. This can be accomplished with a ZDD intersection operation.

- It *satisfies* some clauses. Let $S_{x_i,t}$ be the set of all clauses which contain a literal in $\{x_i, \bar{x}_i\}$ and $x_i = t$ makes this literal true. If these clauses were not yet satisfied, then they become satisfied by this assignment. These clauses are removed from all subsets in the *front* by ZDD existential abstraction.

- It *opens* some clauses. Let $A_{x_i,t}$ be the set of all clauses which were *not activated*, contain a literal in $\{x_i, \bar{x}_i\}$, and $x_i = t$ makes this literal *false*. If instead this literal were assigned true, the clause would be *satisfied* and not *open* and thus not be needed to added to the *front*. All such clauses $A_{x_i,t}$ are added to every subset in the *front* by the ZDD Cartesian product operation.

Note that determining each of these sets depends only on the particular truth assignment to $x_i = t$, and not to the internal state of the *front*. Thus, with each of these sets of clauses, an appropriate action can be taken on the entire *front*. To prune branches from the search containing violated clauses $U_{x_i,t}$, we build the collection $\mathbf{2}^{Clauses \setminus U_{x_i,t}}$ of all sets which do not contain any clauses in $U_{x_i,t}$. This structured collection will have a very small ZDD representation (bounded by $|Clauses|$). We then intersect this collection with the *front*. Finally, it is not hard to see that throughout the algorithm we can remove subsets which are subsumed by some other subset as these correspond to suboptimal partial truth assignments.

Initially, we have no *open* clauses, and the *front* is set to be the collection containing only the empty set, $\mathbf{1}$. For each variable x_i, we modify one copy of the *front* as described above to reflect assigning $x_i = 1$. We modify another copy to reflect assigning $x_i = 0$. Finally, the new *front* is the union of these two, since we must consider promising branches in either case. After all variables are processed, there are two possible outcomes. If there are no branches leading to satisfiability, then the *front* will be empty (equal to $\mathbf{0}$) as it contains sets of *open* clauses, each of which corresponds to a promising branch in the search. If any branches lead to satisfiability, then there will be no open clauses and the *front* will contain the empty

Cassatt(*Vars*, *Clauses*)
  Front $\leftarrow$ **1**
  for $i = 1$ to $|Vars|$ **do**
    front$'$ $\leftarrow$ front
    //Modify front to reflect $x_i = 1$
    Form sets $U_{x_i,1}, S_{x_i,1}, A_{x_i,1}$
      front $\leftarrow$ front $\bigcap \mathbf{2}^{Clauses \setminus U_{x_i,1}}$
      front $\leftarrow$ $\exists$**Abstract**(front, $S_{x_i,1}$)
      front $\leftarrow$ front$\otimes A_{x_i,1}$
    //Modify front' to reflect $x_i = 0$
    Form sets $U_{x_i,0}, S_{x_i,0}, A_{x_i,0}$
      front$'$ $\leftarrow$ front$'$ $\bigcap \mathbf{2}^{Clauses \setminus U_{x_i,0}}$
      front$'$ $\leftarrow$ $\exists$**Abstract**(front$'$, $S_{x_i,0}$)
      front$'$ $\leftarrow$ front$'$ $\otimes A_{x_i,0}$
    //Combine the two branches via Union with Subsumption
    front $\leftarrow$ front $\bigcup_S$ front$'$
  if front = **0 then**
    return Unsatisfiable
  if front = **1 then**
    return Satisfiable

Figure 3: Pseudocode for the Cassatt Algorithm

set (**1**). Since in Cassatt we eliminate subsumed sets, this empty set will subsume all others and the *front* will consist of only the empty set (be equal to **1**).

Pseudocode for the Cassatt algorithm is shown in Figure 3. In general, ZDD algorithms depend heavily on the ordering of ZDD nodes. Also, like most SAT algorithms, Cassatt's performance depends on the order in which variables are processed. To simplify certain steps of the proof as much as possible, it is assumed that indices in the ZDD are ordered according to which clause they represent. Larger clauses appear lower in the ZDD ordering, and among clauses of the same size, clauses which reference variables of lower index appear lower in the variable ordering.

5 Polynomial-Time Refutation of $\overline{PHP_n^{n+1}}$

The Cassatt algorithm is somewhat involved due to its use of ZDD operations; even in the case of these highly symmetric instances the exact number of nodes at a given step is tricky to count. This is primarily due to the node sharing which gives some additional savings in space and runtime at the expense of a conceptually simpler structure. However, in the course of this proof we consider a *partially reduced* ZDD which reveals the underlying structure. Since the ZDD reduction rules eliminate nodes, then the size of this *partially reduced* ZDD is an upper bound on the size of the actual ZDD.

We show here that the number of nodes within Cassatt's main data structure, the *front* is bounded by a polynomial at all steps. This extends to showing that even within ZDD operations, we have a polynomially bounded number of nodes: during ZDD operations, only those nodes which are necessary to produce the resulting ZDD are created. Since intermediate results are hashed, the recursive ZDD procedures only are called once per node (or per pair of nodes) within a given ZDD. If we assume a strong hash function, then, the execution time of every ZDD procedure used in Cassatt is polynomial with respect to the representation size. Because only nodes which are necessary in a given resulting ZDD are produced, and because the procedures execute in time polynomial in the number of nodes, here we focus simply on showing that at each step in the algorithm that the size of the ZDD is polynomially bounded.

Our proof will be structured as follows. First, we will show that the ZDD representation immediately after all variables in some H_k, contains exactly all $n+1-k$ element subsets of these *pigeon* clauses, and will have $(k+1)\cdot(n+1-k)$ nodes. This will give a polynomial bound at regular intervals throughout execution of the algorithm. The remainder of the proof will show that as we consider variables within some H_k, the ZDD in Cassatt does not grow too greatly. Although we will have a bound at regular intervals, we must show a bound between these intervals.

5.1 Bounds After All Variables in H_k

Here we consider that immmediately after completing all variables within some H_k, there is a simple polynomial bound on the number of nodes.

Lemma 2. Let $k \in \{1,2,\ldots,n-1\}$. After completing variable $x_{k(n+1)}$, the *front* contains exactly all $n+1-k$ element subsets of the $n+1$ *pigeon* clauses.

Proof. From the structure of the instances $\overline{PHP_n^{n+1}}$, we know that the only clauses which are in the *cut* are the $n+1$ *pigeon* clauses. Thus the *front* must be composed of subsets of these $n+1$ clauses. From lemma 1, we know that after variable $x_{k(n+1)}$ we may satisfy at most k of these clauses. Then, at least $n+1-k$ of them must remain open. Since we must consider all choices of which k or fewer *pigeon* clauses to satisfy, then all subsets containing $n+1-k$ or more *pigeon* clauses can be in the *front*. Since in Cassatt we eliminate subsumptions, then the smallest subsets (those subsets containing exactly $n+1-k$ elements) subsume all subsets containing more open clauses. As a result, only the $\binom{n+1}{k}$ possible $n+1-k$ element subsets remain.

Lemma 3. Let $k \in \{1,2,\ldots,n\}$. The ZDD representing all k element subsets of n elements contains exactly $(n+1-k)k$ nodes.

Proof. To show this, we first give the form of all such ZDDs in Figure 4.

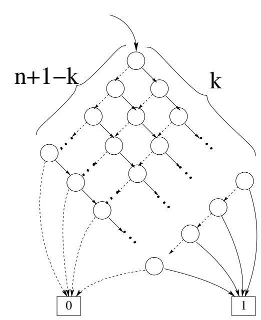

Figure 4: ZDD storing all k element subsets of n elements.

Within this ZDD, in order for a path to reach the terminal **1** node, it must be true for exactly k of the variable values. If less than k values are set to true, the path reaches the **0** node through one of the *E-Child* edges shown on the left of Figure 4. If more than k values are set true, then although it would appear that we follow a path leading to the **1** node, the *Zero-Suppression Rule* implies that we traverse to the **0** node. As a result only those paths which set exactly k variables true will reach the **1** node, and this ZDD stores exactly all k element subsets of an n element set. Since there are a total of n levels, the other 'dimension' of this ZDD is $n+1-k$. Since ZDDs are a canonical representation, whenever it is necessary to store k element subsets we will use exactly $k(n+1-k)$ nodes.

Thus, in Cassatt, when it is necessary to store all $n+1-k$ element subsets from an $n+1$ element set, we use $(n+1-k)(k+1)$ nodes, and the size of the ZDD is polynomially bounded after completing each set of variables H_k.

5.2 Growth Bounds Within Each H_k

Our main goal here is to show that as Cassatt processes variables within each H_k, we do not introduce too many additional nodes. This presented in the following claim.

Claim. Fix $k \in \{0,1,\ldots,n-1\}$. As we process variables within $H_k = \{x_{k(n+1)+1}, x_{k(n+1)+1}, \ldots, x_{(k+1)(n+1)}\}$, the growth of the ZDD is polynomially bounded.

We know before variable $x_{k(n+1)+1}$, that the number of internal ZDD nodes is exactly $(n+1-k)(k+1)$. Similarly, after completing $x_{(k+1)(n+1)}$, the number of internal ZDD nodes is exactly $(n-k)(k+2)$. Then, if we show that during this process the number of nodes is bounded, we will have a bound throughout the execution of the algorithm.

To show this, it is useful to first consider the case when $k \in \{1,2,\ldots,n-2\}$. This differs from the case where $k=0$ since there, the *pigeon* clauses are first activated. It also differs from the case $k=n-1$, since there the *pigeon* clauses lead to conflicts. We can extend the analysis of the cases $k \in \{1,2,\ldots,n-2\}$ to cover these cases without much additional difficulty.

5.2.1 The General Case, $k \in \{1,2,\ldots,n-2\}$

When performing Compressed-BFS over variables within H_k, we would naturally expect the growth of the ZDD to be limited. This is because we know that as we consider variables $H_k = \{x_{k(n+1)+1}, x_{k(n+1)+1}, \ldots, x_{(k+1)(n+1)}\}$ the pairwise exclusion clauses for H_k force at most one of these to be true. As a result, after processing all variables there are essentially only $n+2$ possible branches. If each of these branches led to a polynomially bounded representation, then the entire ZDD would be bounded.

By actually considering how an assignment to some $x \in H_k$ affects the structure of the ZDD, it is possible to show that during the traversal the partially reduced ZDD has a certain regular structure. This structure essentially mimics the heuristic argument given above: there are up to $n+2$ branches and each branch leads to some bounded size ZDD. We will show by induction that the form of this ZDD is maintained throughout all variables within each H_k.

5.2.2 Structure of the *front* During H_k

After variable $x_{k(n+1)+i}$, we expect our internal representation to contain $i+1$ 'branches', each leading to some constraint on the *pigeon* clauses. However this structure is obscured by ZDD node elimination rules. We now consider the case of a partially reduced ZDD to use this structure to bound the number of nodes in the reduced ZDD. We first present the structure of the ZDD after variable $x_{k(n+1)+i}$, then show by induction that this is indeed the structure maintained by the algorithm.

The general structure of the ZDD while processing some variable within H_k is shown in Figure 5. Within this figure, each diamond shaped symbol corresponds to the grid structured ZDD shown in Figure 4, which represents all subsets of a given size. After variable $x_{k(n+1)+i}$, there are i opportunities to branch off from the main path. Each corresponds to setting one variable out of $x_{k(n+1)+1}, x_{k(n+1)+2}, \ldots, x_{k(n+1)+i}$ true, and correspondingly satisfying one *pigeon* clause out of $1,2,\ldots,i$. The constraints each of these branches leads to on the *pigeon* clauses is that at most k of the n remaining *pigeon* clauses (different for each branch) are satisfied, and we have all $n-k$ element subsets of the n remaining

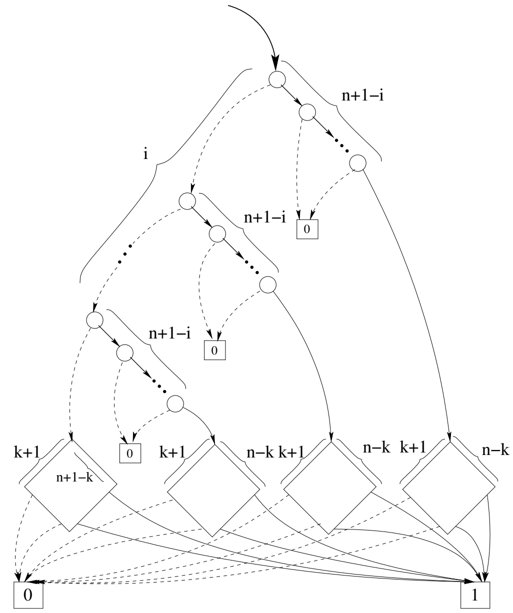

Figure 5: Form of the *front* after $x_{k(n+1)+i}$

clauses at the base of each branch. The leftmost branch corresponds to setting none of $x_{k(n+1)+1}, x_{k(n+1)+2}, \ldots, x_{k(n+1)+i}$ true, and satisfying no *pigeon* clauses with such an assignment. Thus along the leftmost branch, we have all $(n+1-k)$ element subsets just as before. Because of the *pairwise exclusion* clauses, no more than one of $x_{k(n+1)+1}, x_{k(n+1)+2}, \ldots, x_{k(n+1)+i}$ may be true. Along branch $j \leq i$, (corresponding to where $x_{k(n+1)+j}$ is set true), the $n+1-i$ *pairwise exclusion* clauses of the form $(\bar{x}_{k(n+1)+j} + \bar{x}_{k(n+1)+h}), i < h \leq n+1$ remain open along each branch.

We now show by induction that this structure is preserved throughout the operation. Initially, by Lemma 2, the ZDD consists of all $(n+1-k)$ element subsets of the $n+1$ *pigeon* clauses.

Consider setting variable $x_{k(n+1)+1}$ true. Then, we activate n clauses of the form $(\bar{x}_{k(n+1)+1}, \bar{x}_{k(n+1)+j}), 1 < j \leq n+1$, and satisfy the single *pigeon* clause 1. The first step of the algorithm is to remove all branches which contain violated clauses. Since variable $x_{k(n+1)+1}$ does not appear as the end literal for any clause, this step is superfluous. Next, Cassatt existentially abstracts the single satisfied *pigeon* clause, leaving the collection of all $n-k$ element subsets of the remaining n *pigeon* clauses. Finally Cassatt adds the n newly opened *pairwise exclusion* clauses of the form $(\bar{x}_{k(n+1)+1}, \bar{x}_{k(n+1)+j}), 1 < j \leq n+1$ to all sets via Cartesian Product. However, since we ensure that these indices appear above the pigeon clauses in our ZDD ordering, the Cartesian Product operation amounts to simply adding nodes to the top of the ZDD as shown in Figure 6. The resulting structure forms a single branch as shown in Figure 6. Although these operations are themselves complex, within each operation the resulting ZDD is built by a single traversal. This traversal cannot create additional nodes which go unused in the resulting ZDD, and since we know the form of the ZDD both before and after each operation then we can ensure that the size and execution time are bounded during these operations.

Next, consider setting variable $x_{k(n+1)+1}$ false. Here, all clauses which would be activated this step are immediately satisfied. Also, no additional clauses are satisfied or violated. As a result, the resulting ZDD structure is the same as in the previous step: it consists

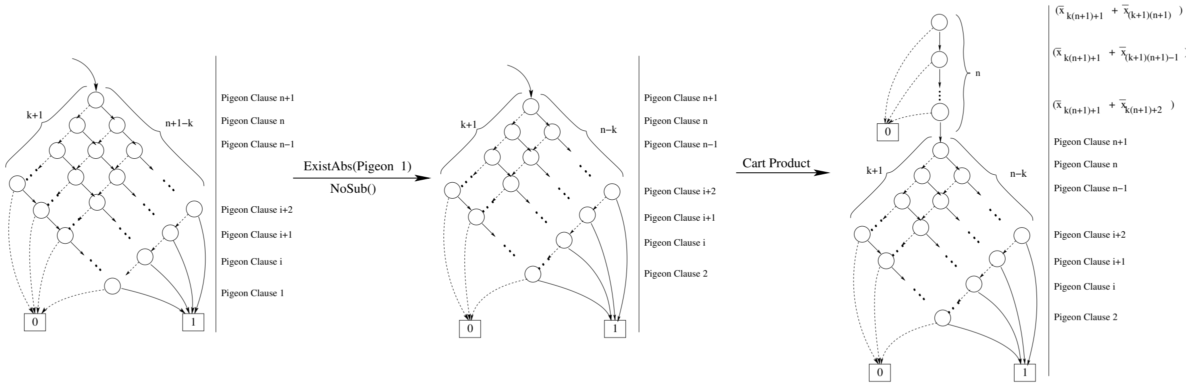

Figure 6: Effects of Setting $x_{k(n+1)+1}$ *true*

of all $n+1-k$ element subsets of the $n+1$ *pigeon* clauses.

Finally both branches are combined via the union with subsumption operation, giving rise to the structure outlined in the previous section. This structure is shown in Figure 7, however here we do not show the merging effects of the ZDD reduction rules, to illustrate the underlying structure of the ZDD.

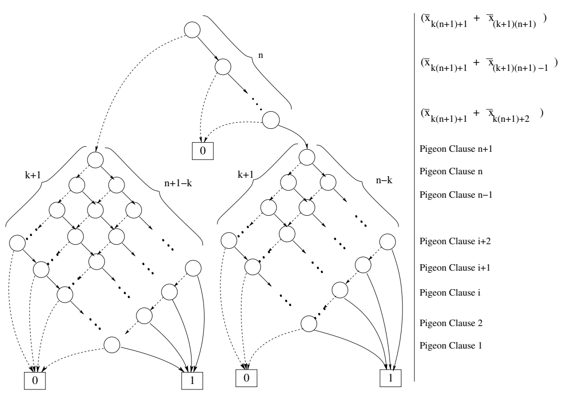

Figure 7: Resulting ZDD after variable $x_{k(n+1)+1}$

Now, assume that after variable $x_{k(n+1)+i}$ we have the precise structure shown in Figure 5 and are processing variable $x_{k(n+1)+i+1}$. If variable $x_{k(n+1)+i+1}$ is set true, we violate the i clauses of the form $(\bar{x}_{k(n+1)+j}+\bar{x}_{(k(n+1)+i+1)})$ where $1 \leq j < i+1$. One of these i clauses appears in each branch of the structure shown in Figure 5. Thus Cassatt will first prune *all* branches except the leftmost branch, whose subsets contains only pigeon clauses.

The key idea in this step is that after eliminating subsets which contain violated clauses, we arrive at the same, grid structured ZDD which appeared after processing variable $x_{k(n+1)}$. This is true in this case since pairwise exclusion clauses constrain more than one of $\{x_{k(n+1)+1},\ldots,x_{k(n+1)+i+1}\}$ from being true. Thus when we consider setting $x_{k(n+1)+i+1}$ true, the only valid branches in the ZDD are those in which all other variables $\{x_{k(n+1)+1},\ldots,x_{k(n+1)+i}\}$ are false, and no additional *pigeon* clauses are satisfied. Since variable $x_{k(n+1)+i+1}$ is set true, it will satisfy the ith *pigeon* clause, and next Cassatt eliminates this clause by existentially abstracting it from the ZDD. Finally, it will open $n-i$ *pairwise exclusion* clauses of the form $(\bar{x}_{k(n+1)+i+1}+\bar{x}_{k(n+1)+j})$ giving rise to a single branch structure similar to the base case $i=1$. This operation is summarized in Figure 8.

A different case occurs when variable $x_{k(n+1)+i+1}$ is set false. In this case no clauses are violated, or activated. Instead, all *pairwise exclusion* clauses of the form $(\bar{x}_{k(n+1)+j}+\bar{x}_{(k(n+1)+i+1)})$ are satisfied. Each branch in the structure shown in Figure 9 except for the leftmost branch contains on of these clauses. Each such clause is removed via Existential Abstraction.

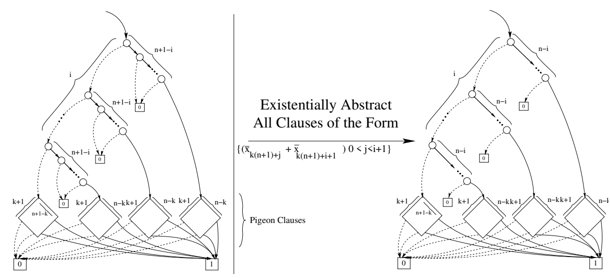

Figure 9: Effects of Setting $x_{k(n+1)+i+1}$ *false*

By combining these two branches via union with subsumption, we effectively increase the number of branches by 1, while removing one node along each branch to obtain exactly the general structure of the ZDD which was introduced in Figure 5. By induction, then, this structure is maintained throughout the progression of the algorithm whenever $k \in \{1,2,\ldots,n-2\}$. Thus, after variable $k(n+1)+i$, the number of ZDD nodes is bounded by $i\,((k+1)(n-k))+(n+2-i))+(k+1)(n+1-k)$, simply by counting nodes in the partially reduced ZDD of Figure 5. Recall that this partially reduced ZDD forms an upper bound on the number of nodes in the *front* since ZDD reduction rules only eliminate nodes from this ZDD.

5.3 Growth Bounds Within H_0

In the analysis of the general case, $k \in \{1,2,\ldots,n-2\}$, we made use of the fact that the *pigeon* clauses were opened previously. Here, the structure of the ZDD is not as complex as in other cases, however, the analysis is similar. It is still useful to consider the unreduced ZDD to highlight the structure present.

The structure of the ZDD after processing variable x_i, where $i \leq n+1$ mimics the general structure shown in Figure 5. However,

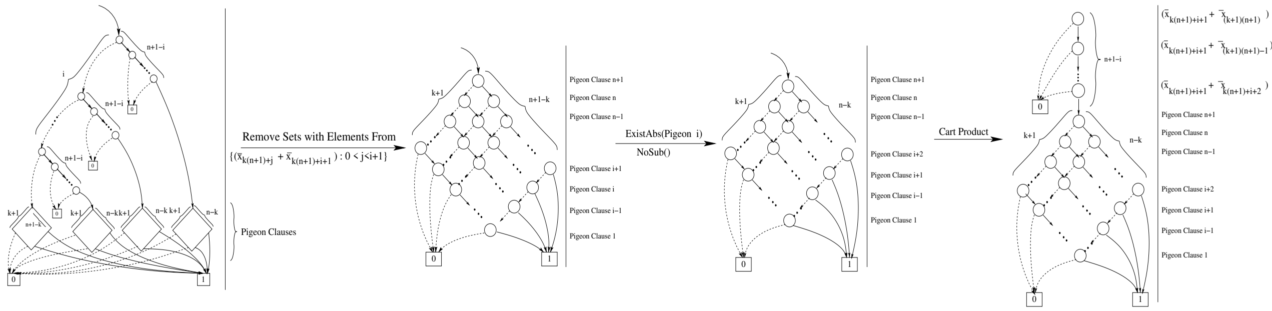

Figure 8: Effects of Setting $x_{k(n+1)+i+1}$ *true*

the grid structured ZDD representing all subsets of a given size is reduced to a degenerate form, representing a single subset of some of these *pigeon* clauses. The number of elements in these subsets is further obscured since at each step, a *pigeon* clause is opened. However, it is not hard to see that after variable x_i, in the leftmost branch (corresponding to setting all $x_1, x_2, \ldots, x_i$ false), we have the single subset containing exactly all *i pigeon* clauses opened thus far. Similarly, at the base of each of the i side branches, we have the single subset containing the remaining $i-1$ *pigeon* clauses, since each such branch corresponds to satisfying one of these clauses.

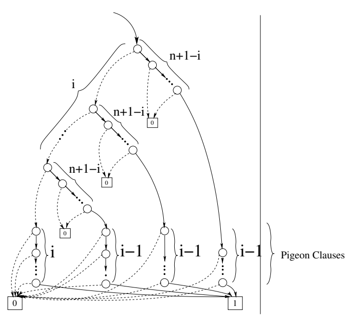

Figure 10: Structure of the ZDD for variables within H_0

To show that this structure is correct, we proceed by induction again. Consider processing variable x_1. When we set x_1 true, we activate *n pairwise exclusion* clauses of the form $(\bar{x}_1 + \bar{x}_j)$ where $1 < j \leq n+1$. When we set x_1 false, we activate the single *pigeon* clause 1. Combining these gives the structure outlined previously.

Next, assume that after variable x_i the partially reduced ZDD has the form shown in Figure 10. When we set x_{i+1} true, we first must prune all branches containing *pairwise exclusion* clauses of the form $(\bar{x}_j + \bar{x}_{i+1})$ where $j < i+1$. By assumption, each branch aside from the leftmost branch in the ZDD must contain one such clause, and after this pruning, the ZDD is reduced to the single set of *i pigeon* clauses along the leftmost branch. Instead of satisfying a *pigeon* clause as in the general case, here we simply do not activate this clause. Finally, we activate $n-i$ *pairwise exclusion* clauses of the form $(\bar{x}_{i+1} + \bar{x}_j)$ where $i+1 < j \leq n+1$.

If we set x_{i+1} false, we violate no clauses, and satisfy all *pairwise exclusion* clauses of the form $(\bar{x}_j + \bar{x}_{i+1})$ where $j < i+1$. Each branch along the ZDD contains one such clause, and it is removed from the ZDD. Finally, this step opens one *pigeon* clause. By the Cartesian product and because the *pigeon* clauses appear lower in the ZDD, this clause is added to the base of each branch in this ZDD. After combining these two together, we arrive at the invariant structure from Figure 10.

Thus, this structure is valid for all variables within H_0, and the size of the corresponding ZDD is bounded by $i((n+1-i)+(i-1))+i = i(n+1)$

5.4 Growth Bounds Within H_{n-1}

Similar to the analysis of H_0, our task is somewhat simplified in this case as we need not consider the ZDD structure of Figure 5. Also, in this case we arrive at conflicts after processing each variable due to the pruning of branches containing violated clauses.

The structure of the ZDD after processing variable $x_{(n-1)(n+1)+i}$, where $i \leq n+1$ again mimics the general structure shown in Figure 5 aside from reuse among different branches. The basic 'branching' of the ZDD is, as mentioned before, a consequence of the *pairwise exclusion* clauses. Thus, as we process variables within any H_i it will arise. However, in this case, each branch leads to the same structure at the base of the ZDD, and there is a total reuse of nodes. The invariant structure (of an unreduced ZDD) at this stage is shown in Figure 11.

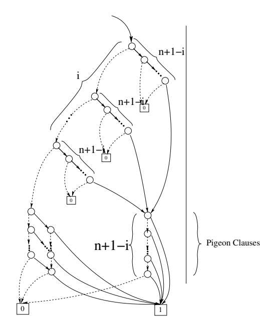

Figure 11: Structure of the ZDD for variables within H_{n-1}

We again proceed by induction to show that this structure is correct. Before the first variable within H_{n-1}, the *front* consists of all 2 element subsets of the $n+1$ *pigeon* clauses. When we consider assigning variable $x_{(n-1)(n+1)+1}$ true, we activate *n pairwise exclusion* clauses, just as in previous steps. Also, we satisfy a single *pigeon* clause. If subsumptions are eliminated, these give rise to a ZDD consisting of all subsets containing every *pairwise exclusion* clause, and exactly one of the remaining *n pigeon* clauses.

When we consider assigning $x_{(n-1)(n+1)+1}$ false, we now violate *pigeon* clause 1, since upon reaching variables in H_{n-1} all *pigeon* clauses become *unit*. Thus, all 2 element subsets containing this *pigeon* clause are eliminated; the resulting ZDD consists of all 2 element subsets from the remaining *n pigeon* clauses. Assigning $x_{(n-1)(n+1)+1}$ false has no other effects, since the *pairwise exclusion* clauses are not opened in this case. The union of these two branches fits within the framework of Figure 11.

Now assume that after processing variable $x_{(n-1)(n+1)+i}$, the *front* is of the form shown in Figure 11. If variable $x_{(n-1)(n+1)+i+1}$ is assigned true, then as before, we violate *pairwise exclusion* clauses appearing in each branch of the ZDD aside from the leftmost branch. Thus after pruning this copy of the *front*, we are left with all 2 element subsets of the remaining $n+1-i$ *pigeon* clauses. One of these *pigeon* clauses is satisfied as a result, and existentially abstracted away, leaving all 1 element subsets of the remaining $n-i$ *pigeon clauses*. Finally the Cartesian product adds $n-i$ *pairwise exclusion* clauses of the form $(\bar{x}_{(n-1)(n+1)+i+1} + \bar{x}_{(n-1)(n+1)+j})$ to each subset, where $i+1 < j \leq (n+1)$. This gives rise to a single new branch of the structure shown in 11.

If variable $x_{(n-1)(n+1)+i+1}$ is assigned false, then we violate the $(i+1)$th *pigeon* clause. Since each branch aside from the leftmost branch allows exactly one of the $n+1-i$ *pigeon* clauses in the cut, then these branches are not pruned completely. Instead, they are updated to reflect allowing exactly one of the remaining $n-i$ *pigeon* clauses. Along the leftmost branch, which ends in all 2 element subsets of the $n+1-i$ *pigeon* clauses, those subsets containing the $(i+1)$th pigeon clause are similarly pruned.

Since the $x_{(n-1)(n+1)+i+1} = 1$ branch in this case contains all 1 element subsets of the remaining $n+1-i$ *pigeon* clauses as a subexpression, then this portion of the ZDD from the $x_{(n-1)(n+1)+i+1} = 0$ branch can be trivially reused when these are combined. This gives rise to the recombination of all side branches in Figure 11. To count the size of the ZDD for variables within H_{n-1} also notice that the portion of the ZDD which represents all 1 element subsets of *pigeon* clauses can be merged with the portion of the ZDD which represents all 2 element subsets (except the topmost node), for additional node savings. Thus, the size of the ZDD for variables within H_{n-1} is bounded by $i(n+1-i)+2(n-i)+1$.

As a result, the number of nodes within the ZDD at any step in the algorithm is bounded polynomially. Then, since the runtime of the algorithm is polynomially dependent on the size of the representation, the runtime is also bounded polynomially. We include a comparison of our bounds predicted value with the actual number of nodes needed to solve the instance `hole-50` in Figure 12. It is

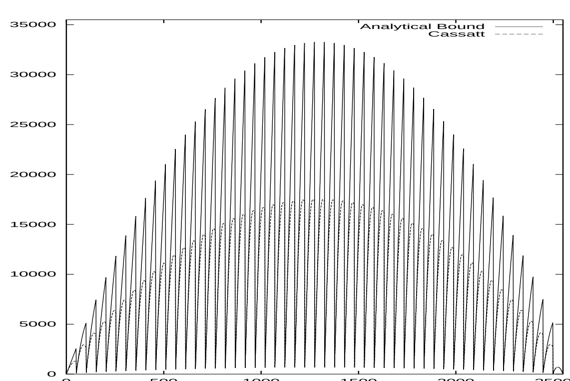

Figure 12: Bounds on the Number of ZDD Nodes

clear that the bound is not tight in most instances as we ignore ZDD reduction rules to simplify the construction. However for variables within H_{n-1}, the bound exactly counts the number of nodes used.

6 Conclusions and Ongoing Work

Our work offers the first analytical proof that the Compressed-BFS search procedure reported in [9] solves SAT instances from the pigeonhole family in polynomial time, thus confirming earlier empirical results. This proof was facilitated by recognizing structural invariants on partially reduced ZDDs within different stages of the algorithm. Once a given structural invariant had been recognized, it was shown correct by induction.

While explicitly formulating this unknown proof system appears difficult due to the complexity of Compressed-BFS procedure, we believe that the details of our polynomiality proof shed some light on it. Namely, one can distinguish several mechanisms which must be reflected in that proof system. First, the steps of proofs may be represented by directed graphs which have exponentially many directed paths (in terms of the number of vertices). Second, those graphs encode Boolean formulas in a compact way by representing elements of Boolean formulas by directed paths. Most importantly, this compact representation facilitates efficient transformations of Boolean formulas.

We conjecture that such graphs can be interpreted as instructions to reuse common Boolean sub-formulas. Therefore, the next step towards formalizing the proof system behind Compressed-BFS may require a description of Compressed-BFS in terms of term rewriting and common sub-formulas.

Our ongoing work proceeds in several directions. First, we are studying modifications of well-known SAT solvers that are required to produce resolution proofs of unsatisfiability rather than just a negative answer. Second, we are trying to modify traces saved by Compressed-BFS so that they form the basis of verifiable proofs. Third, we are trying to further improve our algorithm's runtime by adding Boolean Constraint Propagation to it and see whether this will have any effects related to proof systems.

References

[1] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Comm. ACM*, 5:394–397, 1962.

[2] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.

[3] J. F. Groote and H. Zantema. Resolution and binary decision diagrams cannot simulate each other polynomially. Technical Report UU-CS-2000-14, Utrecht University, 2000.

[4] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[5] J. P. Marques-Silva and K. A. Sakallah. GRASP: A new search algorithm for satisfiability. *ICCAD* , 1996.

[6] S. Minato. Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. *30th ACM/IEEE DAC*, 1993.

[7] A. Mishchenko. An Introduction to Zero-Suppressed Binary Decision Diagrams. http://www.ee.pdx.edu/˜alanmi/research/.

[8] M. Moskewicz et al. Chaff: Engineering an Efficient SAT Solver . *Proc. of IEEE/ACM DAC*, pages 530–535, 2001.

[9] D. B. Motter and I. L. Markov. A compressed breadth-first search for satisfiability. *Proc. ALENEX 2002*, 2002.