Resolution Cannot Polynomially Simulate Compressed-BFS

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

October 1, 2003

Abstract

Many algorithms for Boolean satisfiability (SAT) work within the framework of resolution as a proof system, and thus on unsatisfiable instances they can be viewed as attempting to find proofs by resolution. 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 [1]. Therefore SAT solvers based upon the DLL procedure [2] or the DP procedure [3] must take exponential time. Polynomial-sized proofs of the pigeonhole principle exist for different proof systems, but general-purpose SAT solvers often remain confined to resolution. This result is in correlation with empirical evidence.

Previously, we introduced the Compressed-BFS algorithm to solve the SAT decision problem. In an earlier work [4], 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 tautologies efficiently provable by Compressed-BFS is different than that of any resolution-based procedure.

We hope that the details of our complexity analysis 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. We then use this to show that the overall runtime is bounded by a polynomial.

1 Introduction

Modern high-performance complete SAT solvers such as Chaff [5] and GRASP [6] use the Davis-Logemann-Loveland (DLL) search procedure [2]. DLL is a backtracking algorithm with 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. A recent paper [7] examines the practice of augmenting DLL with clause learning. The authors show that clause learning exponentially improves DLL, but does not overcome the inherent limitations of resolution. Empirical evidence supports this claim. As shown in Figure 1a, the well-known solver zChaff [5], which uses clause learning, empirically takes 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 and solvers consistent with them which may also have practical applications. To this end, we point out that the recently reported Compressed-BFS algorithm [4] empirically solves pigeonhole instances in polynomial time. A plot of runtime on pigeonhole instances for several SAT solvers is shown in Figure 1a. It is evident empirically that the high-performance DLL based solvers cannot

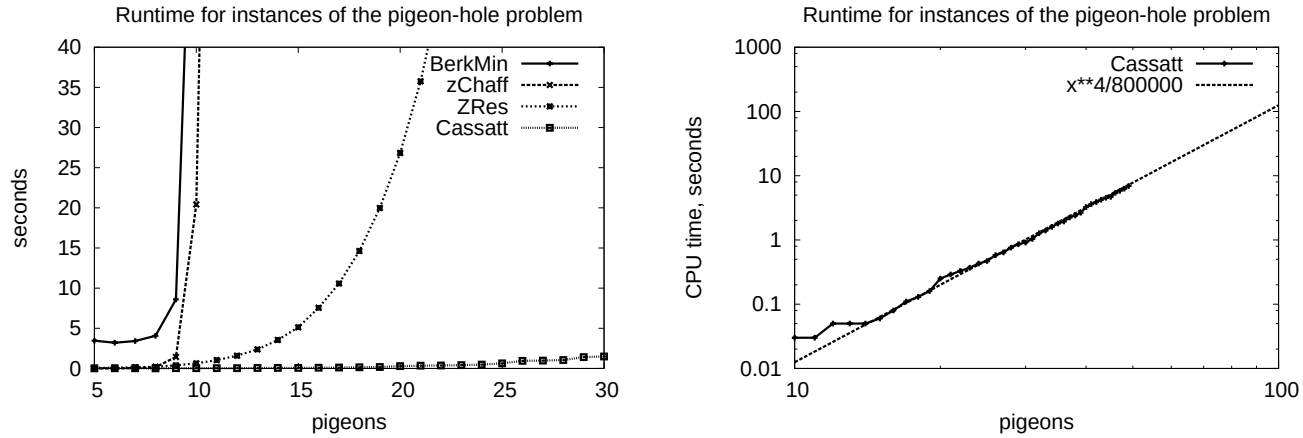

Figure 1: a) Performance of several SAT solvers on `hole-n` instances. b) Asymptotic performance of Compressed-BFS on `hole-n` instances.

solve these instances efficiently while Compressed-BFS is able to quickly show unsatisfiability. In Figure 1b, the runtime of Compressed-BFS on a number of pigeonhole instances is plotted, along with the function $\Theta(x^4)$. Compressed-BFS can show unsatisfiability of $\overline{PHP_{50}^{51}}$ in under 15 seconds on a commodity PC. Since Compressed-BFS is a general-purpose algorithm and competitively solves a number of standard benchmarks (see Appendix A), 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 [4] 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 resolution cannot polynomially simulate the underlying proof system behind Compressed-BFS. While we do not claim resolution is strictly weaker than 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 [8]. 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. In addition, *OBDD-apply* cannot solve pigeonhole instances in polynomial time under any variable ordering, and therefore also cannot polynomially simulate Compressed-BFS [8].

In addition to these theoretical implications, a SAT solver that solves pigeonhole instances in polynomial time can be useful for real-world problems. Within a SAT instance of routing, for example, there can be many embedded $\overline{PHP_n^m}$ instances to enforce the capacities of routing channels [9]. The situation is similar on other structured problems such as planning and scheduling [10]. If a SAT solver cannot efficiently solve these subproblems, it may unnecessarily perform poorly on the problem as a whole [10]. The work in [10] shows how pigeonhole instances can be solved efficiently by using non-clausal learning methods when they are reformulated as instances of 0-1 Integer Linear Programming (ILP). In contrast, our work shows polynomial-time solutions of pigeonhole instances in terms of CNF, where exponential lower bounds on resolution proofs hold.

The attractiveness of combining different techniques such as DLL/DP based SAT approaches and the compression of Binary Decision Diagrams has spawned much research in recent years. Using BDDs to encode the clause database during DLL has been considered [11] as well as the ZRes algorithm, which combines the DP procedure [3] with ZDDs [12]. The ZRes SAT solver [12] empirically solves pigeonhole instances much faster than DLL based solvers as shown in Figure 1, however we are unaware of any published proof of polynomial time complexity on these instances for ZRes. Although ZRes is based on the DP procedure, its state encoding leaves its complexity in this regard an open question. Many other efforts to combine the strengths of multiple approaches (e.g., [13]) or to leverage the power of a compressed data structure (e.g., [14]) have been tried. This idea is likely to be the subject of future research.

Unfortunately, formulating the unknown proof system appears difficult due to the complexity of ZDD algorithms used within the 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. The Compressed-BFS algorithm is described in Section 3. Section 4 introduces the classic pigeonhole instances and some of their relevant properties. In Section 5 we show that the size of the algorithm's main data structure is polynomially bounded. In Section 6 we show that Compressed-BFS proves the pigeonhole principle in polynomial time. In Sections 5 and 6, we give both the general outline of the proof and detailed arguments for each step. Conclusions and our ongoing research are described in Section 7.

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 $t : V \to \{\texttt{true}, \texttt{false}\}$. A *partial truth assignment* for V is a truth assignment to some subset of variables $V' \subseteq V$. A *literal* is a variable or it's negation. A *clause* can be viewed a set of literals. A clause is *satisfied* by a truth assignment t if 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 an open clause are 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 Suppressed Binary Decision Diagram (ZDD). ZDDs can represent combinatorial objects by a set of paths in a Directed Acyclic Graph (DAG) [15, 16, 17]. 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 $\langle X_{index}, X_T, X_E \rangle$ where X_{index} 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 [15, 16, 17, 18, 19], including the union and intersection of two collections of sets, for example.

3 The Compressed-BFS Algorithm

The idea behind the Compressed-BFS algorithm *Cassatt* [1] [4] corresponds directly to a BFS over the tree of partial truth assignments (the two children of a partial assignment are its immediate extensions with 0 and 1). This tree of partial assignments is considered for a given variable ordering. Cassatt implicitly represents partial assignments by their effects on the satisfiability of clauses, and therefore naturally handles symmetric or otherwise equivalent partial assignments. Additionally, Cassatt can identify partial truth assignments that lead to satisfying solutions only if other partial truth assignments to the same set of variables do. This reduces the number of partial truth assignments that must be explored to ensure completeness of the search procedure. As a special case, this includes handling of autark assignments [21], which further differentiates Cassatt from DP or DLL based procedures. Two additional advantages of this algorithm are compressed data representation and the implicit manipulation of large data sets. These are accomplished through the use of ZDD based data structures and relevant algorithms. The two most similar pre-existing algorithms are: (i) ZRes [12], an implementation of the DP procedure using ZDDs, and (ii) an implementation of the DLL procedure with ZDDs [22].

Compressed-BFS 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 $x_{d+1} = \texttt{true}$. It modifies the other copy of the *front* to reflect assigning $x_{d+1} = \texttt{false}$. 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. In this general framework, any data structure which can manipulate collections of sets efficiently can be used in this style of BFS. We chose the ZDD data structure because of its balance between compactness of representation and efficiency of algorithms. 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 $t \in \{\texttt{true}, \texttt{false}\}$) has the following effects on clauses.

[1] Born May 22, 1844, Allegheny City, PA, Mary Cassatt was an American painter and printmaker who exhibited with the Impressionists. [20]

- One or more clauses may be *violated*. 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 [15, 16, 17].

- One or more clauses may be *satisfied*. Let $S_{x_i,t}$ be the set of all clauses which contain a literal in $\{x_i, \overline{x_i}\}$ 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 [17].

- One or more clauses may be *opened*. Let $A_{x_i,t}$ be the set of all clauses which were *not activated*, contain a literal in $\{x_i, \overline{x_i}\}$, 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 [17].

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 (the number of nodes is bounded by $|Clauses|$). We then intersect this collection with the *front*. Also, when considering either CNF instances with empty clauses or clauses with a single variable, this simple taxonomy breaks down. However these cases can be handled with a simple preprocessing step. 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 create two copies of the *front*, and modify one copy of the *front* as described above to reflect assigning $x_i =$ `true`. We modify another copy to reflect assigning $x_i =$ `false`. Finally, the new *front* is the union of these two, since we must consider promising branches in either case. In our implementation, we use the MaxUnion operator, which is built from two additional operators, Maximal and Subsumed Difference for the maintenance of a subsumption-free ZDD [12, 17, 18, 19]. 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 set ($\mathbf{1}$). For a completely worked out instance of Cassatt on a pigeonhole instance, see Appendix B.

Pseudocode for the Cassatt algorithm is shown in Figure 2. 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 with higher index in the ZDD ordering, and among clauses of the same size, clauses which contain variables processed earlier are given higher index in the ZDD ordering.

```
1    Cassatt(Vars, Clauses)
2      front ← 1
3      for i = 1 to |Vars| do
4        front′ ← front
5        //Modify front to reflect x_i = true
6        Form sets U_{x_i,true}, S_{x_i,true}, A_{x_i,true}
7          front ← front ⋂ 2^{Clauses \ U_{x_i,true}}
8          front ← ∃Abstract(front, S_{x_i,true})
9          front ← front ⊗ A_{x_i,true}
10       //Modify front' to reflect x_i = false
11       Form sets U_{x_i,false}, S_{x_i,false}, A_{x_i,false}
12         front′ ← front′ ⋂ 2^{Clauses \ U_{x_i,false}}
13         front′ ← ∃Abstract(front′, S_{x_i,false})
14         front′ ← front′ ⊗ A_{x_i,false}
15       //Combine the two branches via Union with Subsumption
16       front ← front ⋃_S front′
17     if front = 0 then
18       return Unsatisfiable
19     if front = 1 then
20       return Satisfiable
```

Figure 2: Pseudocode for the Cassatt algorithm.

In general, we may ensure that the Cartesian Product operator executes in linear time (in the number of added nodes) by choosing an appropriate node ordering for a given variable ordering. To do this, we would give priority to the criterion that clauses which contain variables processed earlier are given higher index in the ZDD ordering. Then each activated clause would necessarily have lower index than any clause yet appearing in the *front*. However, when this does not occur, the Cartesian Product with a single set can still be performed efficiently by a single traversal as shown in Section 6.

4 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,$x_{i,j}$, each representing that pigeon $1 \leq j \leq n+1$ is in hole

$1 \leq i \leq n$. 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 $x_{1,1}, x_{1,2}, \ldots, x_{1,n+1}$ make up the group H_1, and so on. Then, the group H_i is made up of the variables

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

Encoded this way, the constraints fall into two categories. Within each group of variables H_i, there are $\begin{pmatrix} n+1 \\ 2 \end{pmatrix}$ clauses which we call *pairwise exclusion* clauses of the form

$$(\overline{x}_{i,a} + \overline{x}_{i,b})$$

for $1 \leq a < b \leq n+1$. These clauses prevent more than one pigeon from being in hole i since whenever any one variable $x_{i,a} \in H_i$ is `true`, in order to satisfy all *pairwise exclusion* clauses, all other variables $x_{i,b}$ must be `false`.

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

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

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 $x_{1,j}, \ldots, x_{n,j}$ 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 3. 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 this work, these variables are processed left-to-right, or equivalently, in increasing lexicographic order. Such a variable ordering is easy to find practically, as it corresponds to the ordering which minimizes *cut-width* (as defined by the minimum cut linear arrangement problem [23, 24], which has polynomial time approximations) of the entire instance. A partitioning or placement algorithm can be used to find such an order, if one was not able to explicitly construct it. It is important to note that there is no variable ordering for which a resolution procedure or *OBDD-apply* procedure can refute pigeonhole instances efficiently.

4.1 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*. The *cut* is well-defined graph-theoretic term which can be extended to hypergraphs [25]:

> If X and Y are sets of vertices in a hypergraph H, the set of edges of H with contain vertices from both X and Y is denoted by $[X,Y]$. When $X \cup Y$ is a partition of $V(H)$, the set $[X,Y]$ is called an *edge cut* of H.

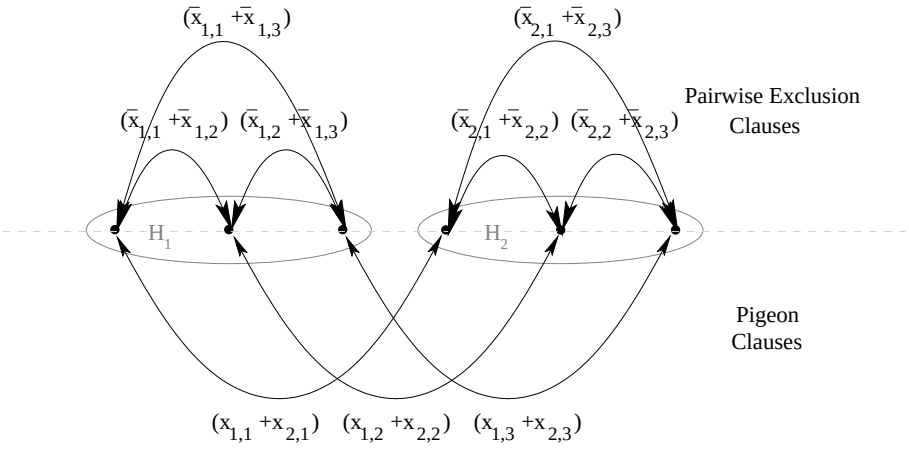

Figure 3: The Instance $\overline{PHP_2^3}$

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 *cut* can be visualized by drawing a vertical line on Figure 3. Then when a partial truth assignment to all variables to the left of the line is being considered, only clauses this line crosses (*cut* clauses) 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 the 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: $n+1$ branches 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 observation is that there are relatively few clauses containing elements in both H_i and H_{i+1}. Figure 4 shows a schematic representation of a larger instance $\overline{PHP_4^5}$ to highlight the general form. As seen in Figure 4, 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 variables in $H_1, \ldots, H_i$, 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 [4]. We can consider all possible truth assignments to variables in $H_1, \ldots, H_i$ to deduce the following lemma.

Lemma 1. Let $k \in \{1, 2, \ldots, n-1\}$. A valid partial truth assignment to variables $x_{1,1}, x_{1,2}, \ldots, x_{k,n+1}$, i.e. all the variables in $H_1, \ldots, H_k$, may satisfy at most k of the $n+1$ *pigeon* clauses.

Proof. First, notice that for $1 \leq i \leq k$ at most one of variables in the set H_i can be assigned `true`. If two or more variables from H_i were `true`, then at least one of the *pairwise exclusion* clauses for H_i

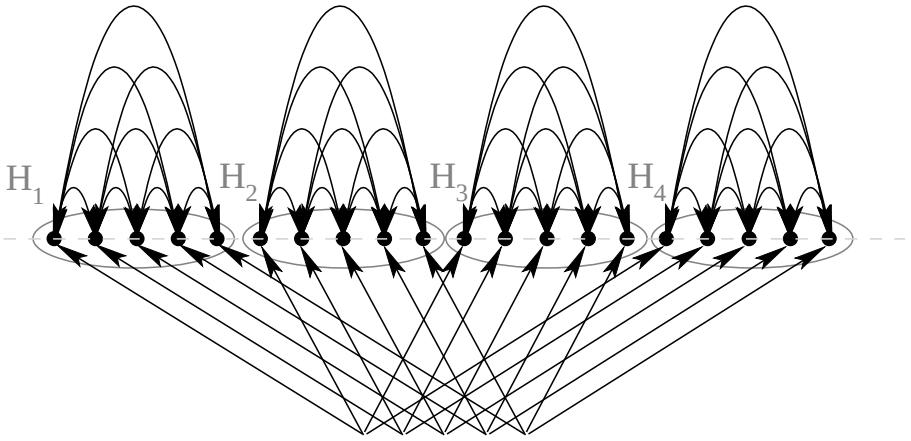

Figure 4: A Larger Instance: $\overline{PHP_4^5}$

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 $H_i = \{x_{i,1}, x_{i,2}, ..., x_{i,n+1}\}$, 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.

■

5 Size Bounds During Refutation of $\overline{PHP_n^{n+1}}$

Determining the exact number of nodes at a given step is cumbersome, and not necessary for an upper bound on the size of the ZDD in Compressed-BFS. 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. Therefore, in the course of this proof we consider a *partially reduced* ZDD which reveals the underlying structure. Since the ZDD reduction rules cannot add 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 Compressed-BFS's main data structure, the *front*, is bounded by a polynomial before and after each individual operation shown in the pseudocode of Figure 2. We will show in the following section that the types of structures we encounter in the refutation of $\overline{PHP_n^{n+1}}$ allow Compressed-BFS to execute in polynomial time, assuming any reasonable hash function.

Our proof will be structured as follows. First, we will show that the ZDD representation immediately after processing all variables in some H_k is the set of $(n+1-k)$-element subsets of *pigeon*

clauses, and will have $(k+1)(n+1-k)$ 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 as well. The techniques used in the proof are valid for the $n![(n+1)!]^n$ 'hole major' variable orderings, but for simplicity the proof assumes that variables are processed in the lexicographic order $x_{1,1}, x_{1,2}, \ldots, x_{1,n+1}, x_{2,1}, \ldots, x_{n,n+1}$.

5.1 Bounds After All Variables in H_k

Here we consider that immediately 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* consists of 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 $x_{k,n+1}$ has been assigned at most k of the *pigeon* clauses may be satsified. 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 will 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 $\begin{pmatrix} n+1 \\ k \end{pmatrix}$ 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 $k(n+1-k)$ nodes.

Proof. To show this, we first give the form of all such ZDDs in Figure 5.

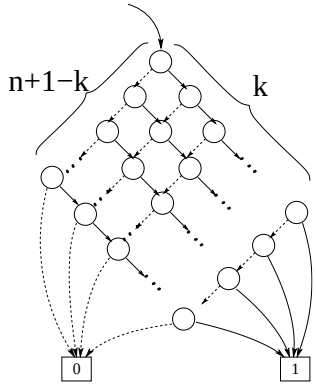

Figure 5: ZDD storing all k-element subsets of n elements.

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

We know that before variable $x_{k,1}$ is processed the number of internal ZDD nodes is exactly $k(n+2-k)$. Similarly, after completing $x_{k,n+1}$, the number of internal ZDD nodes is exactly $(k+1)(n+1-k)$. 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 $k \in \{2,\ldots,n-1\}$. This differs from the case where $k=1$ since there, the *pigeon* clauses are first activated. It also differs from the case $k=n$, since there the *pigeon* clauses lead to conflicts. We can extend the analysis of the cases $k \in \{2,\ldots,n-1\}$ to cover these cases without much additional difficulty.

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

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 within some $H_k = \{x_{k,1},\ldots,x_{k,n+1}\}$ the pairwise exclusion clauses for H_k force at most one of these to be `true`. As a result, after processing variables there are at most $n+2$ possible search "branches". If the ZDD reflects this structure, and each of the corresponding ZDD "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 $x_{k,i}$, 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 $x_{k,i}$, then show by induction that this is indeed the structure maintained by the algorithm.

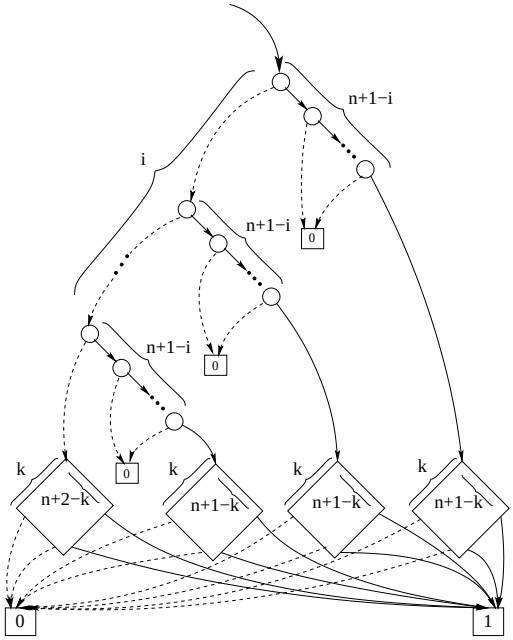

Figure 6: Form of the *front* after $x_{k,i}$.

The general structure of the ZDD while processing some variable within H_k is shown in Figure 6. Within this figure, each diamond shaped symbol corresponds to the grid structured ZDD shown in Figure 5, which represents all subsets of a given size. After variable $x_{k,i}$, there are i opportunities to branch off from the main path. Each corresponds to setting one variable out of $x_{k,1}, x_{k,2}, \ldots, x_{k,i}$ `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 $k-1$ of the n remaining *pigeon* clauses (different for each branch) are satisfied, and we have all $(n+1-k)$-element subsets of the n remaining clauses at the base of each branch. The leftmost branch corresponds to setting none of $x_{k,1}, x_{k,2}, \ldots, x_{k,i}$ `true`, and satisfying no *pigeon* clauses with such an assignment. Thus along the leftmost branch, we have all $(n+2-k)$-element subsets just as before. Because of the *pairwise exclusion* clauses, no more than one of $x_{k,1}, x_{k,2}, \ldots, x_{k,i}$ may be `true`. Along branch $1 \leq j \leq i$, (corresponding to where $x_{k,j}$ is set `true`), the $n+1-i$ *pairwise exclusion* clauses which have the form $(\overline{x}_{k,j} + \overline{x}_{k,h})$, $i < h \leq n+1$ 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 $(n+2-k)$-element subsets of the $n+1$ *pigeon* clauses.

Consider setting variable $x_{k,1}$ `true`. Then, Compressed-BFS activates n clauses of the form $(\bar{x}_{k,1}+\bar{x}_{k,j}), 1 < j \leq n+1$, and satisfies the single *pigeon* clause 1. The first step of the algorithm is to remove all branches which contain violated clauses. Since variable $x_{k,1}$ does not appear as the end literal for any clause, this step is superfluous. Next, Cassatt existentially abstracts the single satisfied *pigeon* clause. In existential abstraction, any occurrence of this clause in any subsets in the *front* will be removed. Consequently, the result of this operation will contain all $(n+2-k)$-element subsets which do not contain *pigeon* clause 1 and all $(n+1-k)$-element subsets not containing *pigeon* clause 1. It is not hard to see this has the form shown in Figure 7, however a simple explanation of this structure is as follows. Similar to Lemma 3, if more than $n+2-k$ inputs are true, the Zero Suppression rule implies this set is not in the collection. Also, if less than $n+1-k$ inputs are true from *pigeon* clauses $\{2,\ldots,n+1\}$, then more than k such inputs are `false` and we traverse to **0** via one of the branches on the left of the graph. Instead, if exactly $n+2-k$ inputs are set `true`, we traverse exactly to the **1** node. Finally, if exactly $n+1-k$ inputs are `true`, we must pass through one of the two bottom-most nodes in Figure 7. If we pass through the left node, then $n-k$ previous inputs have been `true`, and since exactly $n+1-k$ will be `true`, we must pass through the *T-Child* of this node to **1**. If we pass through the right node, then $n+1-k$ previous inputs have been `true`. Whether this input is `false` or `true`, we will have a set with $n+1-k$ or $(n+2-k)$-elements, respectively.

Finally Cassatt adds the n newly opened *pairwise exclusion* clauses which are of the form $(\bar{x}_{k,1}+\bar{x}_{k,j}), 1 < j \leq n+1$, 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 7. The resulting structure forms a single branch as shown in Figure 7.

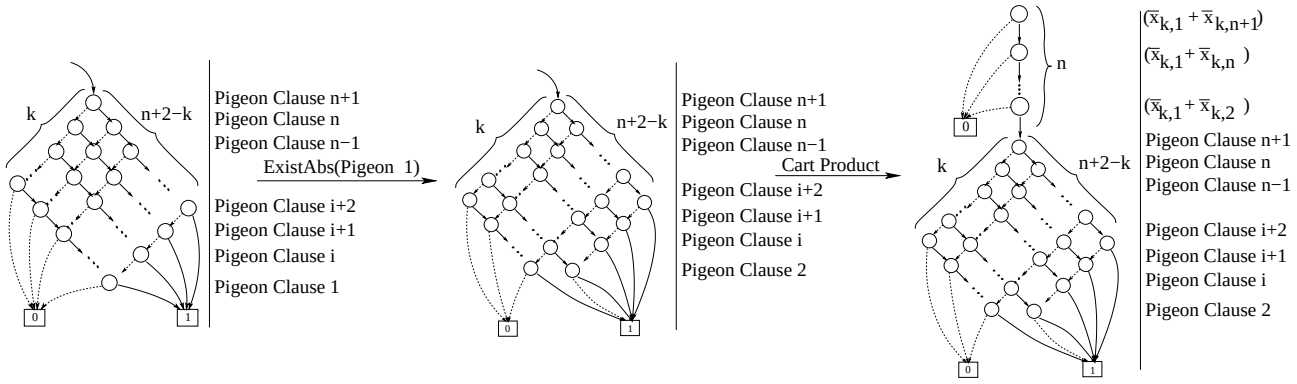

Figure 7: Effects of setting $x_{k,1}$ `true`.

Next, consider setting variable $x_{k,1}$ `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 of all $(n+2-k)$-element subsets of the $n+1$ *pigeon* clauses.

Finally both branches are combined via the subsumption-removing MaxUnion operation, giving rise to the structure outlined in the previous section. This is shown in Figure 8, however here we do not show the merging effects of the ZDD reduction rules, to illustrate the underlying structure of the ZDD. The subgraph of the ZDD containing all $n+2-k$ and $(n+1-k)$-element subsets not containing pigeon clause 1 has subsumptions eliminated from it. As a result, the $(n+1-k)$-element subsets subsume those with $(n+2-k)$-elements, and the resulting ZDD has the form of the grid-structured ZDD of Lemma 3, as shown in 8.

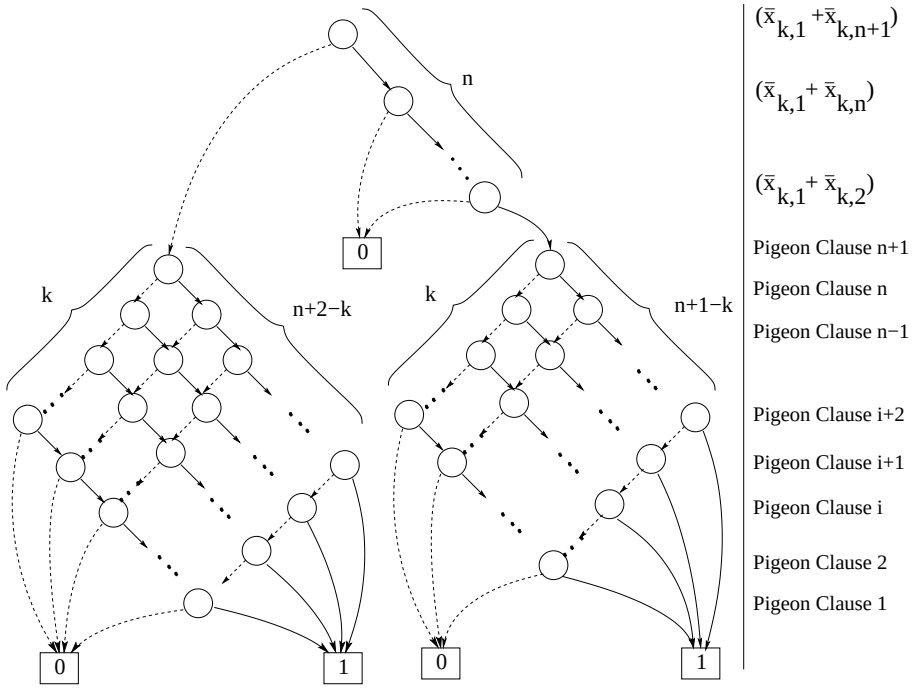

Figure 8: Resulting ZDD after variable $x_{k,1}$.

Now, assume that after variable $x_{k,i}$, $i \in \{1\ldots,n\}$, we have the precise structure shown in Figure 6 and are processing variable $x_{k,i+1}$. If variable $x_{k,i+1}$ is set `true`, we violate the i clauses of the form $(\bar{x}_{k,j} + \bar{x}_{k,i+1})$ where $0 < j < i+1$. One of these i clauses appears in each branch of the structure shown in Figure 6. 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 $x_{k-1,n+1}$. This is true in this case since pairwise exclusion clauses constrain more than one of $\{x_{k,1},\ldots,x_{k,i+1}\}$ from being `true`. Thus when we consider setting $x_{k,i+1}$ `true`, the only valid branches in the ZDD are those in which all other variables $\{x_{k,1},\ldots,x_{k,i}\}$ are `false`, and no additional *pigeon* clauses are satisfied. Since variable $x_{k,i+1}$ is set `true`, it will satisfy the $(i+1)$th *pigeon* clause, and next Cassatt eliminates this clause by existentially abstracting it from the ZDD. The resulting ZDD contains all $(n+2-k)$-element and $(n+1-k)$-element subsets which do not contain *pigeon* clause $i+1$. Finally, it will open

$n-i$ *pairwise exclusion* clauses of the form $(\bar{x}_{k,i+1} + \bar{x}_{k,j})$, $i+1 < j \leq n+1$, giving rise to a single branch structure similar to the base case. This operation is summarized in Figure 9.

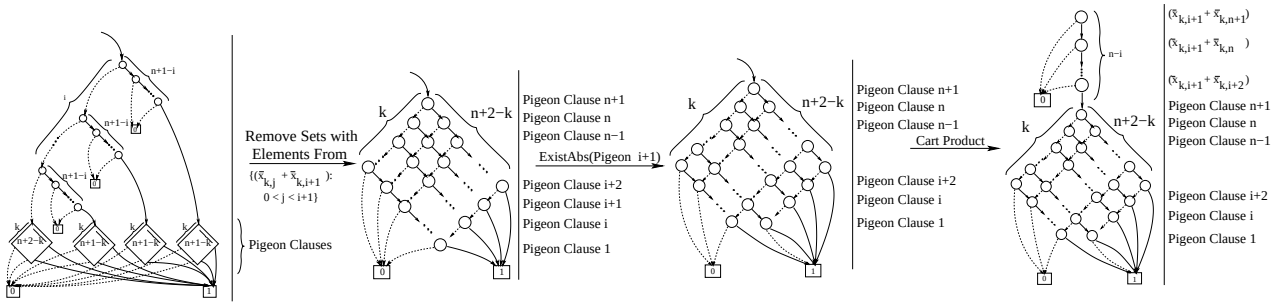

Figure 9: Effects of setting $x_{k,i+1}$ `true`.

A different case occurs when variable $x_{k,i+1}$ is set `false`. In this case no clauses are violated, or activated. Instead, all *pairwise exclusion* clauses of the form $(\bar{x}_{k,j} + \bar{x}_{k,i+1})$, $1 \leq j < i+1$, are satisfied. Each branch in the structure shown in Figure 10 except for the leftmost branch contains one of these clauses. Each such clause is removed via Existential Abstraction.

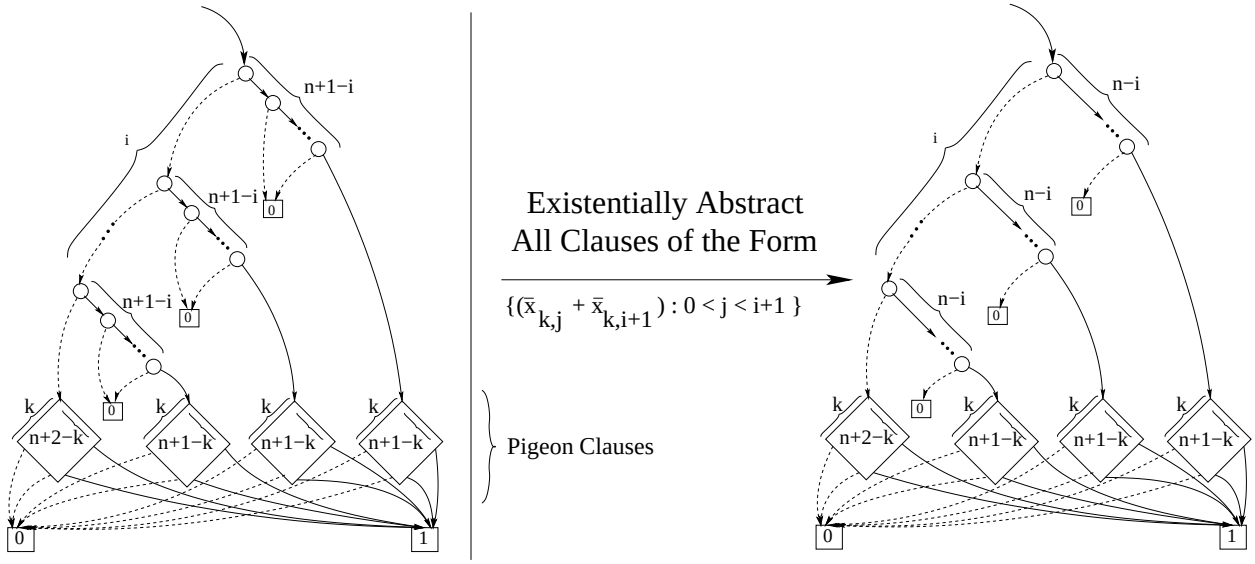

Figure 10: Effects of setting $x_{k,i+1}$ `false`.

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 6. By induction, then, this structure is maintained throughout the progression of the algorithm whenever $k \in \{2, \ldots, n-1\}$. Thus, after variable $x_{k,i}$, the number of

ZDD nodes is bounded by $i(k(n-1-k)+(n+1-i))+k(n+2-k)$, simply by counting nodes in the partially reduced ZDD of Figure 6. 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_1

In the analysis of the general case, $k \in \{2,\ldots,n-1\}$, we made use of the fact that the *pigeon* clauses were opened previously. When processing a variable $x_{1,i}$ where $1 \leq i \leq n+1$, 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 $x_{1,i}$, where $1 \leq i \leq n+1$ mimics the general structure shown in Figure 6. However, 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 $x_{1,i}$, in the leftmost branch (corresponding to setting all $x_{1,1}, x_{1,2}, \ldots, x_{1,i}$ `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.

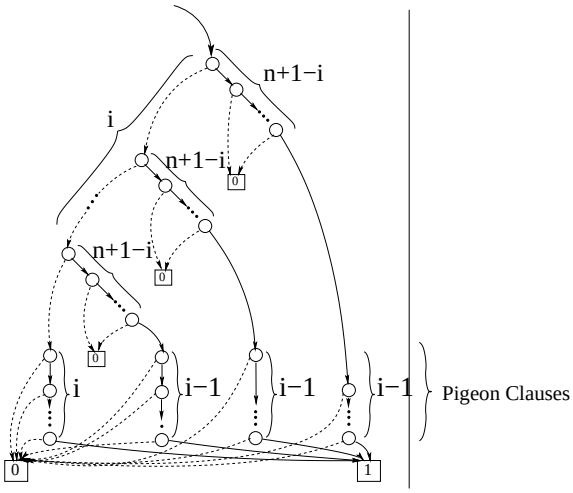

Figure 11: Structure of the ZDD for variables within H_1.

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

Next, assume that after variable $x_{1,i}$ the partially reduced ZDD has the form shown in Figure 11. When we set $x_{1,i+1}$ `true`, we first must prune all branches containing *pairwise exclusion* clauses of the form $(\bar{x}_{1,j}+\bar{x}_{1,i+1})$ where $1 \leq j < i+1$. 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 $(\bar{x}_{1,i+1}+\bar{x}_{1,j})$ where $i+1<j\leq n+1$.

If we set $x_{1,i+1}$ `false`, we violate no clauses, and satisfy all *pairwise exclusion* clauses of the form $(\bar{x}_{1,j}+\bar{x}_{1,i+1})$ where $1\leq j<i+1$. 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 11.

Thus, this structure is valid for all variables within H_1, 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

Similar to the analysis of H_1, our task is somewhat simplified in this case as we need not consider the ZDD structure of Figure 6. 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 $x_{n,i}$, $1\leq i\leq n+1$, again mimics the general structure shown in Figure 6 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 12.

We again proceed by induction to show that this structure is correct. Before the first variable within H_n, the *front* consists of all 2-element subsets of the $n+1$ *pigeon* clauses. When we consider assigning variable $x_{n,1}$ `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 $x_{n,1}$ `false`, we now violate *pigeon* clause 1, since upon reaching variables in H_n 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 $x_{n,1}$ `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 12.

Now assume that after processing variable $x_{n,i}$, $i\in\{1,\ldots,n\}$, the *front* is of the form shown in Figure 12. If variable $x_{n,i+1}$ 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 $(\bar{x}_{n,i+1}+\bar{x}_{n,j})$ to each subset, where $i+1<j\leq n+1$. This gives rise to a single new branch of the structure shown in Figure 12.

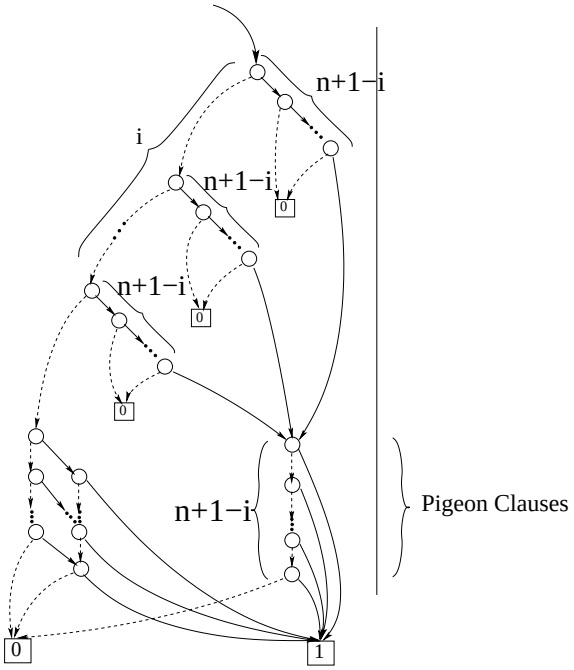

Figure 12: Structure of the ZDD for variables within H_n.

If variable $x_{n,i+1}$ 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 $x_{n,i+1} =$ `true` 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 $x_{n,i+1} =$ `false` branch can be trivially reused when these are combined. This gives rise to the recombination of all side branches in Figure 12. To count the size of the ZDD for variables within H_n 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 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. We include a comparison of our bounds predicted value with the actual number of nodes needed to solve the instance `hole-50` in Figure 13. It is 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, the bound exactly counts the number of nodes used.

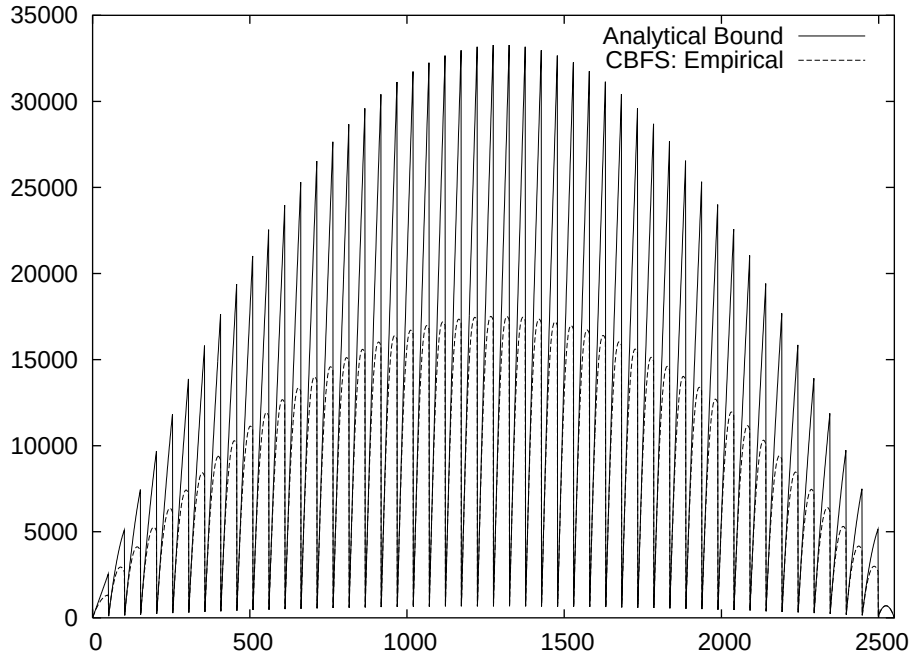

Figure 13: Bounds on the number of ZDD nodes.

6 Polynomial-time Refutation of $\overline{PHP_n^{n+1}}$

During the refutation of $\overline{PHP_n^{n+1}}$, the number of nodes in the *front* is polynomially bounded after processing each variable. In addition, we noted explicitly in most cases that before and after each ZDD operation the number of nodes is similarly bounded; this was necessary to construct the explicit bound after each variable. The remaining cases will be discussed as necessary.

In general, ZDD operations are often performed as traversals over the ZDD or a pair of ZDDs and for such traversals, one can bound the amount of work done by the number of nodes in the argument ZDDs with appropriate caching of results [15]. For example, the ZDD intersection operation $f \cap g$ can create at most $O(|f| \cdot |g|)$ additional nodes. In addition, if we perfectly cache the results of function calls, the ZDD intersection $f \bigcap g$ cannot cause more than $O(|f| \cdot |g|)$ different function calls. Since the number of nodes is polynomially bounded, we assume any reasonable hash function which does not cause such traversals to require superpolynomial time.

In Compressed-BFS, however, some ZDD operations are not such traversals. These ZDD operations use the results of computations recursively, and are not *a priori* bounded by the size of their arguments. However, we will show that even these operations perform in polynomial time when the ZDD assumes any of structures shown in the previous section. To show this, we will first give pseudocode for the nontraversal operations used in Compressed-BFS.

It is clear that the ZDD intersection performs in polynomial time. As mentioned previously, if a suitable ordering for the ZDD nodes is chosen, the Cartesian Product operation can also be performed

```
1   Product(ZDD f, ZDD-Set s)
2       if(f_index < s_index)
3           return ⟨f_index, Product(f_T, s), Product(f_E, s)⟩
4       if(f_index > s_index)
5           return ⟨s_index, Product(f, s_T), 0⟩
6       // f_index ≠ s_index
```

Figure 14: Pseudocode for Cartesian product with a single set.

quickly (linear in the number of activated clauses). Although this ordering can always be selected, for the purposes of this proof we instead chose a node ordering which highlighted the significance of branches within the ZDD. Regardless of node ordering, in Cassatt we need only form the Cartesian product of a single set to the main ZDD, which can be performed in a single traversal as shown in Figure 14, and thus this operation will execute in polynomial time. In this pseudocode the ZDD s must contain a single set of clauses, none of which may be contained in any of the sets in f.

The remaining operations are Existential Abstraction, and MaxUnion [26, 17]. These operators both use additional routines recursively within their definition, and the runtime is not as simple to bound. To consider these operations, we will again need to do a case-by-case analysis based on the ZDD structures introduced in the previous section.

6.1 Time Complexity Bounds Within H_k

Consider the general case of processing a variable $x_{k,i}$ in H_k, where $k \in \{2, \ldots, n-1\}$. Then the ZDD has the form shown in Figure 6. Here we show the runtimes of both the Existential Abstraction routine and the MaxUnion routine while processing this variable are polynomially bounded. To do this, a detailed pseudocode for each routine is given and discussed within this general case. The remaining cases of variables within H_1 and H_n follow almost trivially.

6.1.1 Existential Abstraction

The Existential Abstraction operation (Figure 16) can be written recursively by using the standard ZDD Union operation as a subroutine (Figure 15) [17, 27]. However, since existential abstraction depends on the result of this union, in general we can not easily bound its complexity in terms of input size and thus it is not readily apparent that in the general case this operation will take time polynomial in $|f|$ and $|g|$.

Claim. When Existential Abstraction is used in the refutation of $\overline{PHP_n^{n+1}}$, the ZDD has a specific structure for which the operation will execute in polynomial time.

In Compressed-BFS, the Existential Abstraction routine is used in two cases. First, when a variable is set `true`, we may need to existentially abstract a single *pigeon* clause. However, when only a

```
1    Union(ZDD f, ZDD g)
2      if(f == 0) return g
3      if(g == 0) return f
4      if(f == g) return f
5      if(f_index < g_index)
6        return ⟨f_index, Union(f_E, g), f_T⟩
7      if(f_index > g_index)
8        return ⟨g_index, Union(g_E, f), g_T⟩
9      if(f_index == g_index)
10       return ⟨f_index, Union(f_E, g_E), Union(f_T, g_T)⟩
```

Figure 15: Pseudocode for ZDD Union.

```
1    ExistAbs(ZDD f, ZDD-Set s)
2      if(f_index > s_index)
3        return ExistAbs(f, s_T)
4      if(f_index < s_index)
5        return ⟨f_index, ExistAbs(f_T, s), ExistAbs(f_E, s)⟩
6      if(f_index = s_index)
7        return Union(ExistAbs(f_T, s_T), ExistAbs(f_E, s_T))
```

Figure 16: Pseudocode for ZDD Existential Abstraction.

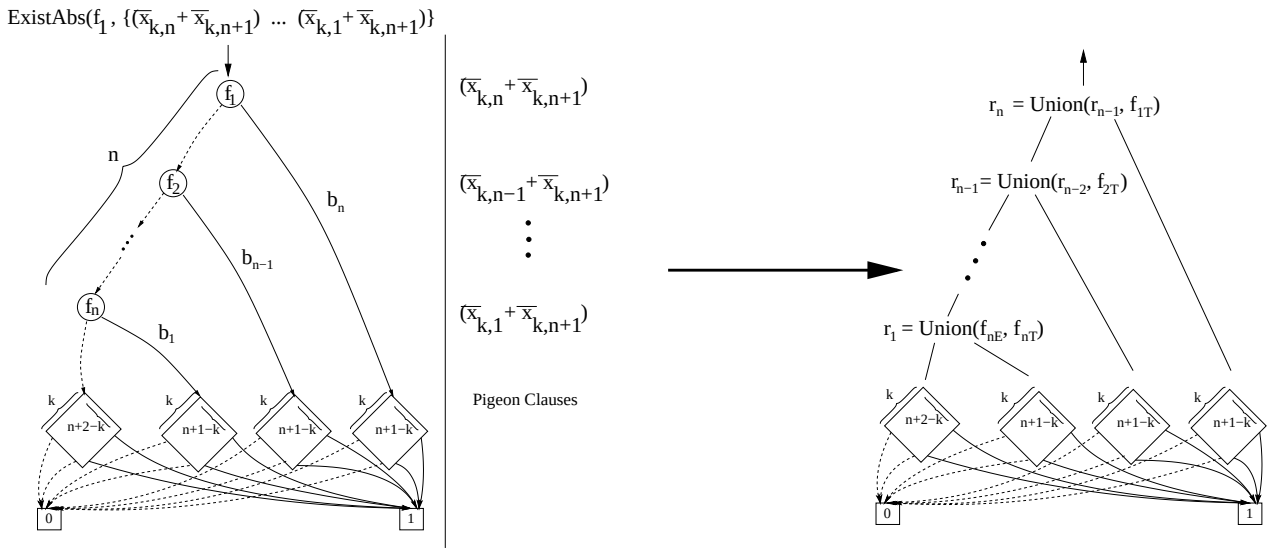

Figure 17: Effects of Existential Abstraction when setting $x_{k,n+1}$ `false`.

single clause must be abstracted, the operation is equivalent to finding the union of the two cofactors of this variable. It is clear that in this case, the complexity is bounded polynomially.

Secondly, when a variable is set `false`, we must abstract i pairwise exclusion clauses. We will now show this case is also bounded polynomially. When we process all but the last variable in H_k, each branch of the main ZDD will contain one such clause, as described previously. Existential abstraction of these clauses then recurses along each branch. However, because the *E-Child* of each node corresponding to a *pairwise exclusion* clause is **0**, we effectively attempt to form the union of the remaining ZDD with **0**. In these cases, the union step of existential abstraction performs no additional work, and the operation effectively boils down to a single pass over the ZDD.

Finally, a different case occurs when we process the last variable $x_{k,n+1}$ in some H_k. At this step, the nodes to be existentially abstracted are the nodes which separate the ZDD into branches, and their *E-Child* is not zero. However each node along this main branch will be removed. As a result, the operation boils down to forming the union of each of the $n+1$ branches as shown in Figure 17. The main branch holds all $(n+2-k)$-element subsets of the $n+1$ pigeon clauses, while each remaining branch b_j holds $(n+1-k)$-element subsets of the n pigeon clauses other than pigeon clause j. Instead of decomposing the operation further, we now focus on the result r_i of each union in Figure 17.

Lemma 4. Each result ZDD r_i contains $O(n^2)$ nodes.

Proof. We show this by giving the explicit form of each r_i in figure 18. The union r_1 of the main branch and the first branch, b_1, will contain all $(n+2-k)$-element subsets as well as those $(n+1-k)$-element subsets which do not contain *pigeon* clause 1. It is not hard to verify this ZDD has the form shown in Figure 18: similarly to Lemma 3, if more than $n+2-k$ inputs are `true`, the Zero Suppression rule implies this set is not in the collection. Also, if less than $n+1-k$ inputs are `true` from *pigeon* clauses $\{1,\ldots,n\}$, then more than k such inputs are `false` and we traverse to **0** via one

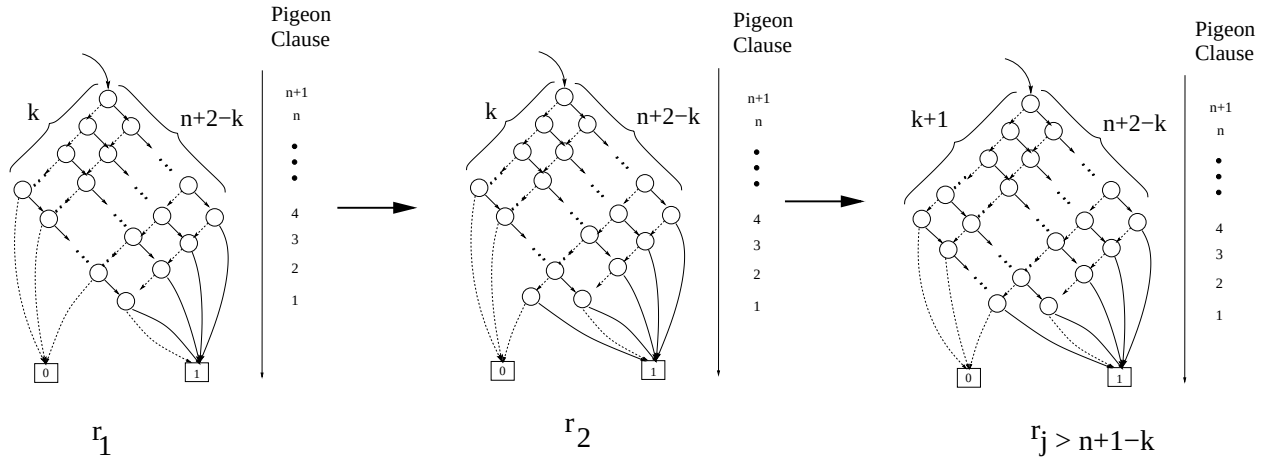

Figure 18: ZDD after Union r_i during Existential Abstraction.

of the branches on the left of the graph. Instead, if exactly $n+2-k$ inputs are set `true`, we traverse exactly to the **1** node. Finally, if exactly $n+1-k$ inputs from $\{1,2,\ldots,n\}$ are `true`, we must pass through the bottom-most node at level 1. If the input 1 is `true`, we should traverse to **1** as we have a set with $(n+2-k)$ elements. Otherwise, we should also traverse to **1** since we have a set with $(n+1-k)$-elements, not containing *pigeon* clause 1. It follows that this ZDD is the result r_1 of the union between the main branch and b_1.

The remaining unions are similarly shown to have the form in Figure 18. Assume that after branch b_i we have the form of Figure 18, and are performing the union with branch b_{i+1}. Then the only subsets which need to be added are those $(n+1-k)$-element subsets (not containing $i+1$) which contain all elements in $\{1,\ldots,i\}$. This is the case because the structure in Figure 18 already contains all $(n+1-k)$-element subsets which do not contain all elements in $\{1,\ldots,i\}$ and hence contains those such subsets not containing $i+1$. If it is the case that there are no such subsets (i.e. if $i > n+1-k$ then no $(n+1-k)$-element subset can contain all required i elements) then the structure is unchanged. However, if there are $(n+1-k)$-element subsets which do not contain $i+1$, and do contain all of $\{1,\ldots,i\}$, then these subsets may be added by creating a single node as indicated in Figure 18. This is the case since in order for an $n+1-k$ element subset to contain all of $\{1,\ldots,i\}$ it must traverse the *T-Child* for each of those nodes. The only portion of the previous ZDD which did not allow such a traversal to reach **1** is augmented with an additional node at level i.

■

Since each union operation's runtime is bounded by the product of its inputs' sizes, the entire Existential Abstraction routine will execute in polynomial time. It follows that during the execution of Compressed-BFS during variables within some H_k for $k \in \{2,\ldots,n-1\}$ that the Existential Abstraction procedure takes polynomial time in n, as it performs successive unions on ZDDs of bounded size.

```
1   operator\S (ZDD f, ZDD g)
2     if(f = 0 ||g = 1||g = f) return 0
3     if(f = 1||g = 0) return f
4     if(f_index > g_index)
5       return f\_S g_E
6     if(f_index < g_index)
7       return ⟨f_index, f_T\_S g, f_E\_S g⟩
8     if(f_index == g_index)
9       return ⟨f_index, (f_T\_S g_T)\_S g_E, f_E\_S g_E⟩
```

Figure 19: Pseudocode for ZDD Subsumed Difference.

```
1   Maximal(ZDD f)
2     if(f = 1||f = 0) return f
3     if(f_T = f_E) return f_T
4     let A = Maximal(f_E)
5     return ⟨f_index, Maximal(f_T)\_S A, A)⟩
```

Figure 20: Pseudocode for ZDD Subsumption Elimination.

6.1.2 Union with Subsumption Removal

After the front is modified to reflect assigning a given variable `true`, and a copy of the front is modified to reflect assigning a given variable `false`, these two copies must be combined into a single data structure. In our implementation, this is accomplished by using a subsumption-removing union operator, MaxUnion, to facilitate maintaining a subsumption-free database of clauses [26, 17]. In previous steps of the proof, it was noted that it was possible for some sets to subsume others. The subsumption-removing union operator removes such sets while combining the two branches.

The subsumption-removing union operator MaxUnion is built on two other ZDD procedures. The first operator is the subsumed-difference operator $\backslash_S$ [26, 18, 19]. In Figure 19 we list pseudocode for this operator, note that there are alternate ways of implementing it [26]. $A\backslash_S B$ returns a ZDD containing all of the sets contained in A that are not subsumed by some set contained in B. The second procedure is Maximal, whose pseudocode is listed in Figure 20, removes all subsumed sets from the given ZDD.

To see that performing the MaxUnion to combine the two possibilities after some variable $x_{k,i}$ in H_k executes in polynomial time, recall the structure reached after setting such a variable $x_{k,i}$ true in Figure 9 and the structure reached after setting such a variable false in Figure 10. These two structures

```
1    MaxUnion(ZDD f, ZDD g)
2       if(f = 0) return Maximal(g)
3       if(g = 0) return Maximal(f)
4       if(f = g) return Maximal(f)
5       if(f = 1||g = 1) return 1
6       if(f_index < g_index)
7          return ⟨f_index, MaxUnion(f_E, g), Maximal(f_T)\_S MaxUnion(f_E, g)⟩
8       if(f_index > g_index)
9          return ⟨g_index, MaxUnion(g_E, f), Maximal(g_T\_S MaxUnion(g_E, f)⟩
10      if(f_index = g_index)
11         return ⟨f_index, MaxUnion(f_T, g_T)\_S MaxUnion(f_E, g_E), MaxUnion(f_E, g_E)⟩
```

Figure 21: Pseudocode for ZDD Subsumption-free Union.

must be combined via MaxUnion. When $i < n$, the newly added pairwise exclusion clauses in Figure 9 have lower index (thus appearing higher in the figure) than any other clauses in these structures, and this is where the MaxUnion(f, g) begins. Then the MaxUnion recurses (based on line 7). Since $f_E = \mathbf{0}$, the recursion branch to find MaxUnion(f_E, g) simply returns Maximal(g). Thus, the effect of MaxUnion is to find the Maximal of both branches as shown in Figure 22. It will then combine these branches via Subsumed Difference.

Within each branch, some subsumed sets may be present due to the existential abstraction operation used. Note that the intersection operation can only remove sets, and the Cartesian product adds elements to every set, so neither of these operations can create subsumed sets.

The $x_{k,i} = \texttt{true}$ branch will then have subsumptions as it contains all $(n+1-k)$-element sets not containing *pigeon* clause *i* as well as all $(n+2-k)$-element sets not containing pigeon clause *i*. It follows that these $(n+1-k)$-element sets will subsume the larger sets, leaving only all $(n+1-k)$-element sets not containing pigeon clause *i*, and thus the action of the Maximal operator on the ZDD is nontrivial.

However, it is possible to trace the execution of this operator over the ZDD due to its regular structure. The execution in this step is essentially the same regardless of the implementation of the Subsumed Difference operator since all necessary subsumptions are performed by line 3 of the Maximal routine (Figure 20).

As the Maximal operator proceeds in a bottom-up fashion, when it views higher nodes in the ZDD, the subsumptions present in lower portions will already have been eliminated. Then whenever Maximal returns, the ZDD beneath that point has already taken its final form: all $(n+1-k)$-element subsets. However, the Maximal operator will also perform a subsumed difference operation in order to ensure the completeness of the search. We will now show that this subsumed difference operation essentially performs no useful task in this case, and runs in time polynomial in n.

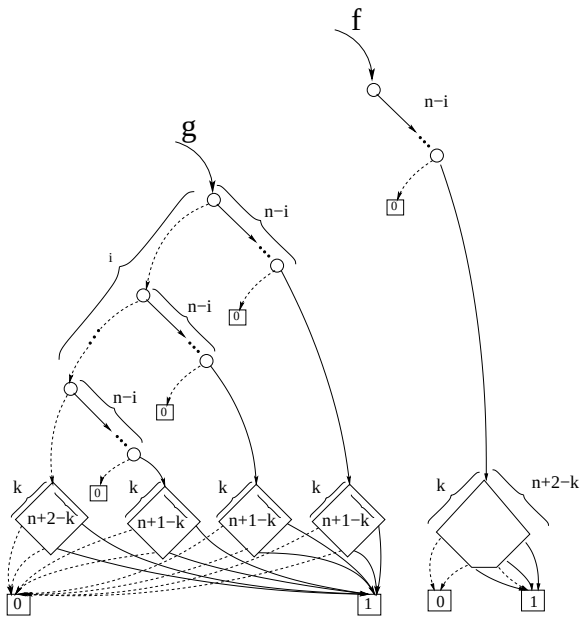

Figure 22: ZDDs f, g to be combined via MaxUnion.

Lemma 5. Consider two nodes A and B on the same level of a ZDD containing all k-element subsets out of n elements. Suppose that A appears to the "left" of B in the sense that to reach A from the least common ancestor of A and B we traverse more *T-Children* than to reach B, then the Subsumed Difference $A \backslash_S B$ returns A and executes in polynomial time.

Proof. First notice that A consists of all i-element sets and B consists of all j-element sets, where $i < j \leq k$. This is true since the sub-portion of the ZDD represented by both A and B has the structure of Lemma 3. Also, since A is "left" of B, there are fewer remaining *T-Children* to traverse, implying via Lemma 3 that A contains subsets of fewer elements than B. Then the Subsumed Difference will return A as no subset of B can subsume a subset of A.

To show that the Subsumed Difference $A \backslash_S B$ does not create additional nodes, we can proceed by induction. First, since nodes A and B have the same index, we always recurse based on line 9 of the pseudocode given for Subsumed Difference. For the base case, we have A and B as the only two nodes on the $n-1$th level of an n level ZDD of all subsets. Then $A_T = \mathbf{1}$ so $A_T \backslash_S B_T = \mathbf{1}$ as $B_T \neq 1$. Similarly, based on the terminal cases given in the pseudocode, $(A_T \backslash_S B_T) \backslash_S B_E = \mathbf{1}$ as well. Finally since $B_E = \mathbf{0}$ we have $A_E \backslash_S B_E = A_E$. It follows from line 9 of the routine that we return the node $\langle A_{index}, \mathbf{1}, A_E \rangle$ which is precisely A, and no additional nodes need to be created.

Now assume that $A \backslash_S B$ does not create additional nodes when A, B have the same index in a ZDD of all k-element subsets, and A is left of B. Then there are three cases.

- Case 1: $A_T = \mathbf{1}$. In this case, $A_T \backslash_S B_T = \mathbf{1}$ as it is impossible that $B_T = \mathbf{1}$. Then

$(A_T \setminus_S B_T) \setminus_S B_E = \mathbf{1}$ as well. Finally $A_E \setminus_S B_E$ is formed. If $B_E = \mathbf{0}$, then we will return the node $\langle A_{index}, \mathbf{1}, A_E \rangle$ which is precisely A. Otherwise, we will recursively evaluate the subsumed difference $A_E \setminus_S B_E$. But A_E is "left" of B_E, and by assumption this evaluation creates no additional nodes and returns A_E. Then we still return the node $\langle A_{index}, \mathbf{1}, A_E \rangle = A$ and thus this step creates no additional nodes.

- Case 2: $B_E = \mathbf{0}$ and $A_T \neq \mathbf{1}$. In this case, we form $A_T \setminus_S B_T = A_T$ without creating additional nodes by assumption. Then, $(A_T \setminus_S B_T) \setminus_S B_E = A_T$ without additional nodes simply since $B_E = \mathbf{0}$. Finally, we evaluate $A_E \setminus_S B_E = A_E$ without creating additional nodes since $B_E = \mathbf{0}$. Then we return $\langle A_{index}, A_T, A_E \rangle = A$ and again no additional nodes are necessary.

- Case 3: $A_T \neq \mathbf{1}$ and $B_E \neq \mathbf{0}$. Then the three subsumed difference operations $A_T \setminus_S B_T = A_T$, $(A_T \setminus_S B_T) \setminus_S B_E = A_T$, and $A_E \setminus_S B_E = A_E$ can each be performed without creating additional nodes by assumption. We then return the node $\langle A_{index}, A_T, A_E \rangle = A$ and create no additional nodes.

It follows that in all cases, when we evaluate $A \setminus_S B$, we do not create additional nodes. Since $A \setminus_S B$ does not create additional nodes and only depends recursively on subsumed differences of elements of the same index, there are a limited number of calls which can be made. Namely, if results of these calls are hashed, there are a total of at most $O(n)$ nodes on a given level, and correspondingly $O(n^2)$ calls can be made per level. Then the evaluation of any $A \setminus_S B$ satisfying the conditions of the Lemma will execute time polynomial in n. Further the total time complexity of finding all such $A \setminus_S B$ within a given ZDD storing k-element sets is bounded by a polynomial in n.

■

Since the Subsumed Difference is called after the recursive evaluation of Maximal (Figure 20, line 5), we have exactly the conditions of Lemma 5 for each call to Subsumed Difference. Since the total time of all such calls is bounded by a polynomial in n, it follows that the time complexity of execution of Maximal on the $x_{k,i} = \texttt{true}$ branch will take time bounded by a polynomial.

The $x_{k,i} = \texttt{false}$ branch will have no subsumptions unless we are processing the last variable in H_k, since in these other cases, the only nodes existentially abstracted away are pairwise exclusion clauses from each branch of the ZDD. When only these clauses are removed, no new subsumed sets are created, and all previously existing subsumed sets were removed during the last MaxUnion. In addition, when processing the $x_{k,i} = \texttt{false}$ branch, we recurse to find the Maximal of $i+1$ grid structured ZDDs forming subgraphs of the main ZDD. Since each of these $i+1$ Maximal operations will take polynomial time, the entire Maximal will execute in polynomial time.

However, when we process the last variable $x_{k,n+1}$ in some H_k then we arrive at a structure similar to that in the $x_{k,i} = \texttt{true}$ branch. Namely, we have all $(n+1-k)$-element subsets of the $n+1$ pigeon clauses, as well as all $(n+2-k)$-element subsets of these clauses. Then all $(n+1-k)$-element subsets should subsume the larger sets. However there are no activated clauses in the $x_{k,n+1} = \texttt{true}$ branch to partition the action of MaxUnion into two Maximal operations as in the previous case.

Recall that the $x_{k,n+1} = \texttt{true}$ branch contains all $(n+1-k)$-element and $(n+2-k)$-element subsets which do not contain *pigeon* clause $n+1$. It follows then, that this ZDD is entirely contained

within the $x_{k,n+1} = \texttt{false}$ branch. If at the topmost node f of the $x_{k,n+1} = \texttt{false}$ branch, we traverse along the *E-Child*, then we restrict to all $n+1-k$ and $(n+2-k)$-element sets to those not containing *pigeon* clause $n+1$, *exactly* the same ZDD as the $x_{k,n+1} = \texttt{true}$ branch.

The action of MaxUnion on these ZDDs appears somewhat unnecessary as one is entirely contained as a subset of the other. It will now be shown that MaxUnion captures this relationship and effectively only performs a Maximal operation on the entire ZDD.

When MaxUnion(f, g) is applied to these two branches, since their root nodes are at different levels, the rule in line 7 is applied. Thus we first attempt to find MaxUnion(f_E, g). However, since $f_E = g$, we attempt to find MaxUnion(g, g) which reduces to Maximal(g). Then as described in the analysis of $x_{k,i} = \texttt{true}$, the action of Maximal on a structure of this form runs in polynomial time.

Next, MaxUnion(f, g) recursively calls Maximal(f_T). However, Maximal can execute on this structure in polynomial time. Finally, MaxUnion(f, g) performs the subsumed difference Maximal(f_T)$\backslash_S$ Maximal(g). However, this exactly satisfies the conditions of Lemma 5, and thus executes in polynomial time.

It follows that the MaxUnion of the two branches can be performed in polynomial time while processing any variable within H_k where $k \in \{2, \ldots, n-1\}$.

6.2 Time Complexity Bounds Within H_1

The structure assumed when we examine a variable of the form $x_{1,i}$ where $1 \leq i < n+1$ is significantly simpler than that of the general case. It maintains the general "branching" however, and nearly all of the analysis from the previous section remains valid in this case as well. In particular, the action of the Existential Abstraction is exactly the same.

The MaxUnion operation again reduces to two Maximal calls when $1 \leq i < n+1$. However in these simplified cases there are no subsumptions to eliminate. It is clear that Maximal requires polynomial time when its argument contains a single set: the *E-Child* of each node is **0**. Since as before, the action of Maximal on the branching structure of the ZDD is time bounded, then the time complexity of MaxUnion is also bounded.

Finally when processing variable $x_{1,n+1}$ the resulting ZDD structure is slightly different. Recall that for variables in H_k where $k \in \{2, \ldots n-1\}$ we have that the structure for the $x_{k,n+1} = \texttt{true}$ branch was entirely contained within the $x_{k,n+1} = \texttt{false}$ branch. This is because when performing Existential Abstraction, we form unions $r_1, r_2, \ldots r_n$. All Union results r_j, where $n+2-k \leq j \leq n$, are the same, and hold all $(n+1-k)$-element subsets as well as all $(n+2-k)$-element subsets of the $n+1$ pigeon clauses. In this case, there are no such Unions r_j, as the last union formed is r_n and $k = 1$. Then the structure holds all $(n+2-k = n+1)$-element subsets, as well as all n-element subsets which contain *pigeon* clause 1. Thus the MaxUnion is between this collection and the collection for the $x_{k,n+1} = \texttt{true}$ branch: the single n-element set not containing the *pigeon* clause 1.

Although this case is different, the presence of the *pigeon* clause 1 in subsets for the $x_{k,n+1} = \texttt{false}$ branch causes the MaxUnion operation to again partition into two Maximal operations followed by a Subsumed Difference. Again due to the structure of the ZDD, these will be polynomially bounded, by Lemma 5 and that finding Maximal of a ZDD containing a single subset is efficient.

It follows that all operations for variables within H_1 execute in polynomial time.

6.3 Time Complexity Bounds Within H_n

In this final case, we consider processing a variable $x_{n,i}$ where the initial structure is of the form shown in Figure 12. The execution of Existential Abstraction over this structure is bounded by the same arguments as before. For the case of variable $x_{n,n+1}$, however, both the $x_{n,n+1} = \texttt{true}$ and $x_{n,n+1} = \texttt{false}$ branches are identically $\mathbf{0}$.

Similarly, the MaxUnion is partitioned when $1 \leq i < n+1$ and trivial when $i = n+1$. Then all operations throughout the algorithm execute in polynomial time during the refutation of $\overline{PHP_n^{n+1}}$.

7 Conclusions and Ongoing Work

Our work offers a detailed analysis that shows the Compressed-BFS search procedure reported in [4] solves SAT instances from the pigeonhole family in polynomial time, 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. The time bound on ZDD operations was accomplished by noting the exact effects of each ZDD operation.

While explicitly formulating this unknown proof system appears difficult due to the complexity of the ZDD algorithms used during the 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. Another natural direction of research is to determine whether the addition of pruning based on Boolean Constraint Propagation will affect the efficiency of Compressed-BFS. However our preliminary investigations hint that this type of idea will not have as dramatic effects as in resolution-based procedures. Future directions of research also include explicitly formulating hard examples for Compressed-BFS.

References

[1] A. Haken, "The Intractability of Resolution," *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985.

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

[3] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *Journal of the ACM*, vol. 7, pp. 201–215, 1960.

[4] D. Motter and I. Markov, "A Compressed Breadth-First Search for Satisfiability," *Proc. 4th Workshop on Algorithm Engineering and Experiments*, 2002.

[5] M. Moskewicz *et al.*, "Chaff: Engineering an Efficient SAT Solver," *Proc. of IEEE/ACM Design Automation Conferenec*, pp. 530–535, 2001.

[6] J. Marques-Silva and K. Sakallah, "GRASP: A New Search Algorithm for Satisfiability," *Proc. Intl. Conf. Computer-Aided Design*, 1996.

[7] P. Beame, H. Kautz, and A. Sabharwal, "Understanding the Power of Clause Learning," in *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-2003)*, Morgan Kaufman, 2003.

[8] J. Groote and H. Zantema, "Resolution and Binary Decision Diagrams Cannot Simulate Each Other Polynomially," Tech. Rep. UU-CS-2000-14, Utrecht University, 2000.

[9] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry," *IEEE Trans. on CAD*, vol. 22(9), pp. 1117–1137, Sept 2003.

[10] H. Dixon and M. Ginsberg, "Inference Methods for a Pseudo-Boolean Satifiability Solver," *The Eighteenth National Conference on Artificial Intelligence (AAAI-2002)*, 2002.

[11] F. Aloul, I. Markov, and K. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *Proc. Intl. Conf. Computer-Aided Design*, 2001.

[12] P. Chatalic and L. Simon, "ZRes: the old DP meets ZBDDs," *Proc. of the 17th Conf. of Autom. Deduction (CADE)*, 2000.

[13] P. Bjesse, G. Andersson, B. Cook, and Z. Hanna, "A Proof Engine Approach to Solving Combinational Design Automation Problems," *Proc IEEE/ACM Design Automation Conference*, 2002.

[14] P. Ferragina and G. Manzini, "An Experimental Study of a Compressed Index," *Proc. 12th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2001.

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

[16] S. Minato, *Representations of Discrete Functions*, pp. 1–26. Kluwer Academic Publishers, 1996.

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

[18] O. Coudert, "Two-Level Logic Minimization: an Overview," *Integration, the VLSI Jour.*, vol. 17, pp. 97–140, 1994.

[19] O. Coudert, "Solving Graph Optimization Problems with ZBDDs," *Proceedings of the 1997 European Design and Test Conference*, 1997.

[20] Mary Cassatt at http://www.ibiblio.org/wm/paint/auth/cassatt/

[21] O. Kullmann, "Investigations on Autark Assignments," *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, vol. 107, 2000.

[22] F. Aloul, M. Mneimneh, and K. Sakallah, "Backtrack Search Using ZBDDs," *Intl. Workshop on Logic and Synthesis, (IWLS)*, 2001.

[23] M. Garey and D. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 2000.

[24] V. Kann, "A compendium of NP optimization problems," http://www.nada.kth.se/ viggo/wwwcompendium/node56.html

[25] J. Bondy, "Basic Graph Theory: Paths and Circuits," *Handbook of Combinatorics*, vol. I, p. 34, 1995.

[26] P. Chatalic and L. Simon, "Multi-Resolution on Compressed Sets of Clauses," *Proc. of 12th International Conference on Tools with Artificial Intelligence (ICTAI-2000)*, Nov. 2000.

[27] A. Mishchenko, "EXTRA v. 1.3: Software Library Extending CUDD Package: Release 2.3.x." http://www.ee.pdx.edu/˜alanmi/research/extra.htm

[28] L. Simon, D. LeBerre, and E. Hirsch, "The SAT2002 Competition," http://www.satlive.org/SATCompetition/2002/onlinereport/index.html

Appendix A: Cassatt on Other Benchmarks

Table 1 shows how Cassatt and a few well known SAT solvers fare on a subset of benchmarks taken from the SAT02 competition [28]. The benchmarks were run on machines with 2.0 GHz processors with 1GB of RAM. "# Solved" is the number of benchmarks which the solver was able to complete

	Bart Series		Lisa Series		Homer Series	
	# Solved	Time	# Solved	Time	# Solved	Time
Cassatt	21	1585.4	0	50400	**15 of 15**	**4.26**
BerkMin	21	80.58	11	23334	6	32837
mChaff	2	70375	10	19249	12	14788
zChaff	5	58079	**12 of 14**	**12273**	6	35575
GRASP	**21 of 21**	**1.33**	0	50400	0	54000
ZRes	0	75600	0	50400	1	53891
	Ca Series		Dp Series		XOR-Chain Series	
	# Solved	Time	# Solved	Time	# Solved	Time
Cassatt	3	19935	0	79200	**27 of 27**	**30.4**
BerkMin	8	41.78	**21 of 22**	**5829.0**	6	75876
mChaff	**8 of 8**	**2.95**	18	14826	18	53983
zChaff	8	6.90	18	16482	20	50677
GRASP	7	5843.4	12	36184	0	97200
ZRes	6	7659.2	7	54987	27	104.19

Table 1: A comparison of Cassatt with other SAT solvers on difficult benchmarks from the SAT 2002 competition [28]. Each solver was given 3600 CPU seconds per benchmark. The number of benchmarks completed within that time as well as the total CPU time for each suite of benchmarks is shown.

given a timeout of 3600 seconds per benchmark. "Time" is the total amount of time taken for all of the benchmarks in a series of benchmarks. Solvers which could not finish a benchmark within the timeout period of 3600 seconds were charged 3600 seconds for that particular benchmark.

These benchmarks show that Cassatt performs competitively on a collection of difficult benchmarks. In fact the XOR-Chain series of benchmarks, on which Cassatt peforms quite well, contains the smallest unsatisfiable benchmark that was unsolved in the SAT02 competition [28]. The two other series where Cassatt does well, the Bart series and Homer series, represent FPGA switch-box problems as described in [9].

Appendix B: Cassatt Example – Refutation of $\overline{PHP_2^3}$

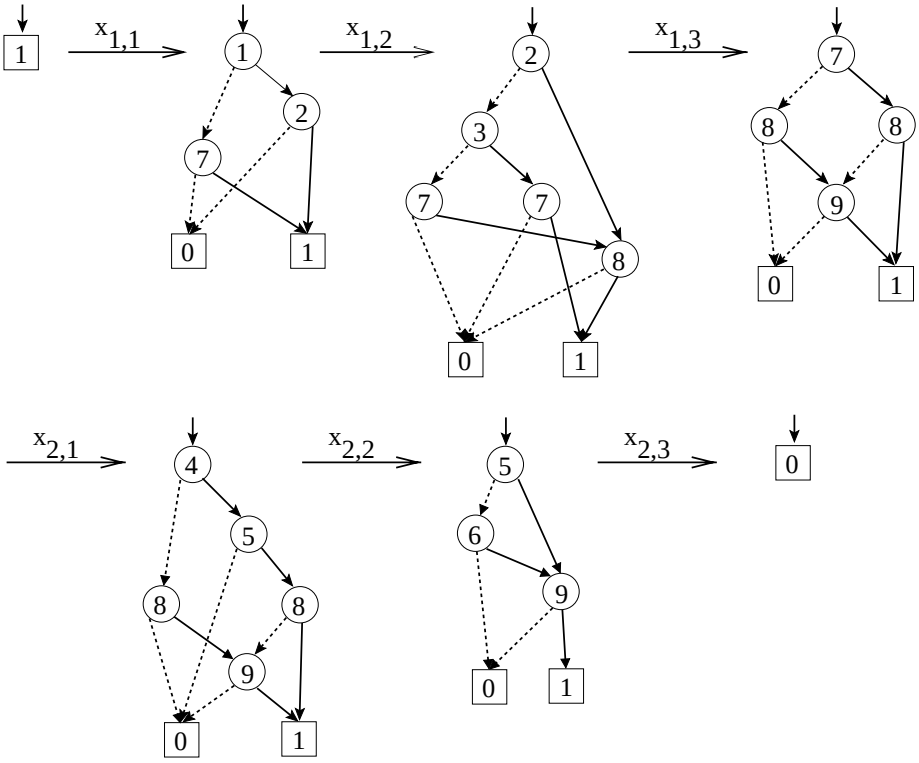

Figure 23: Progression of the *front* during refutation of $\overline{PHP_2^3}$.

The SAT instance of $\overline{PHP_2^3}$ contains 6 variables, $\{x_{1,1}, x_{1,2}, x_{1,3}, x_{2,1}, x_{2,2}, x_{2,3}\}$, and 9 clauses $\{(\bar{x}_{1,1}+\bar{x}_{1,2})$, $(\bar{x}_{1,1}+\bar{x}_{1,3})$, $(\bar{x}_{1,2}+\bar{x}_{1,3})$, $(\bar{x}_{2,1}+\bar{x}_{2,2})$, $(\bar{x}_{2,1}+\bar{x}_{2,3})$, $(\bar{x}_{2,2}+\bar{x}_{2,3})$, $(x_{1,1}+x_{2,1})$, $(x_{1,2}+x_{2,2})$,$(x_{1,3}+x_{2,3})\}$ (numbered 1-9 respectively). See Figure 23 for the progression of the ZDD representation of the *front*.

Cassatt begins by setting the front to $\{\{\}\}$. Next Cassatt processes $x_{1,1}$. If $x_{1,1}$ is set `true`, clauses 1 and 2 are opened and clause 7 is satisfied. If $x_{1,1}$ is set `false`, clauses 1 and 2 are satisfied and

clause 7 is opened. Thus the *front* becomes $\{\{1,2\},\{7\}\}$.

Next $x_{1,2}$ is processed. If $x_{1,2}$ is set `true`, clause 1 is violated, clause 3 is opened, and clause 8 is satisfied. If $x_{1,2}$ is set `false`, clauses 1 and 3 are satisfied and clause 8 is opened. Thus the *front* becomes $\{\{3,7\},\{2,8\},\{7,8\}\}$.

Next $x_{1,3}$ is processed. If $x_{1,3}$ is set `true`, clauses 2 and 3 are violated and clause 9 is satisfied. If $x_{1,3}$ is set `false`, clauses 2 and 3 are satisfied and clause 9 is opened. Thus the *front* becomes $\{\{7,8\},\{7,9\},\{8,9\},\{7,8,9\}\}$. $\{7,8,9\}$ is subsumed by other elements of the *front*, so the *front* is reduced to $\{\{7,8\},\{7,9\},\{8,9\}\}$.

Next $x_{2,1}$ is processed. If $x_{2,1}$ is set `true`, clauses 4 and 5 are opened and clause 7 is satisfied. If $x_{2,1}$ is set `false`, clauses 4 and 5 are satisfied and clause 7 is violated. Thus the *front* becomes $\{\{4,5,8\},\{4,5,9\},\{4,5,8,9\},\{8,9\}\}$. $\{4,5,8,9\}$ is subsumed by other elements of the *front*, so the *front* is reduced to $\{\{4,5,8\},\{4,5,9\},\{8,9\}\}$.

Next $x_{2,2}$ is processed. If $x_{2,2}$ is set `true`, clause 4 is violated, clause 6 is opened, and clause 8 is satisfied. If $x_{2,2}$ is set `false`, clauses 4 and 6 are satisfied and clause 8 is violated. Thus the *front* becomes $\{\{5,9\},\{6,9\}\}$.

Lastly $x_{2,3}$ is processed. If $x_{2,3}$ is set `true`, clauses 5 and 6 are violated and clause 9 is satisfied. If $x_{2,3}$ is set `false`, clauses 5 and 6 are satisfied and clause 9 is violated. Thus the *front* becomes $\{\}$, which means that $\overline{PHP_2^3}$ is unsatisfiable.