The Open University of Israel
Department of Mathematics and Computer Science

Structure Analysis in Boolean Functional Synthesis

Thesis submitted as partial fulfillment of the requirements
towards an M.Sc. degree in Computer Science
The Open University of Israel
Department of Mathematics and Computer Science

By
Ziv Avissar

Prepared under the supervision of **Dr. Dror Fried**

June 2023

Abstract

Boolean Functional Synthesis (BFnS) is the problem of synthesizing a Boolean function from a Boolean specification that describes a relation between input and output variables. Due to the many applications of BFnS, such as in circuit design, QBF solving, and reactive synthesis, efforts are continuously made to explore and better study BFnS. In this work we deepen our understanding of BFnS by analyzing underlying graph structures of BFnS instances. This is motivated by a chain of works on SAT instances, where analysis of graph features, such as graph modularity, is used to explore the performance of SAT solvers on industrial SAT instances. We first show that unlike instances that are random, industrial BFnS instances admit high modularity, even more than is common in SAT. Observing that, we construct a novel BFnS random instance generator with controlled modularity, which we use to study the effect of modularity on performance of state-of-the-art BFnS solvers. Finally we white-box these solvers to determine how the structure of generated instances changes iteratively through the solving process. Our findings indicate that instances modularity has a direct effect on the various solvers performance and their behavior.

Acknowledgements

I would like to thank the colleagues who've been crucial and helpful in this work.
To Lucas M. Tabajara for help kicking off this work as well as many discussions along the way, and guidance on working with Back and Forth
To Supratik Chakraborty for many discussions along the way.
And to his group of students, and especially Shetal Shah, for crucial brainstorming discussions and much guidance on working with their tool, BFSS.
To Kuldeep S. Meel and Priyanka Golia for fruitful discussions as well as much guidance and help working with their tool, Manthan.
To Danny Berend for useful discussions.

To Dror Fried my thanks are given for a lengthy process and the determination to follow through. This thesis is the culmination of a long research which has taken us from highs to lows, and having a supervisor whose curiosity rivals mine own has been instrumental in finishing it.

Finally, To my parents and family for fanning the ember of this work through laborious years in the army.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Boolean Functional Synthesis (BFnS) is the problem of extracting a boolean function from a given Boolean specification that describes a relation between input and output variables. Due to its clean formulation, BFnS has found applications in many areas, among which circuit design [24], QBF solving [30], reactive synthesis [35], and automated program synthesis [33]. As such, study of BFnS has evolved over the last decade and various research groups have suggested different approaches and tools to battle BFnS (e.g. [2, 19, 22, 23, 34]). Even so, BFnS still remains a very hard problem to solve at a large scale, which is not surprising as it is a computational complexity challenging problem, specifically Π_2^P-hard. Moreover, it is very likely that not all BFnS instances even admit a polynomial size solution [1].

One of the means to battle BFnS is to reason on the structure of the given specification. There are already a few stratagems for solving BFnS such as factorization [23] and sequential decomposition [14] that utilize decomposition of the specification into clauses. A question to ask, however, is whether there is a more inherent structure of BFnS specifications that can be related to solutions run-time by the various BFnS solvers. This challenge is already met for the problem of satisfiability (SAT) that also consists of input as a Boolean formula. Indeed, discerning structure from a given Boolean formula has been explored in SAT in recent years [6]. Specifically, it was shown that unlike random Boolean formulas, real-world originated Boolean formulas, sometimes called industrial benchmarks, admit certain graph properties when represented as graphs in various ways. This resulted in a whole line of research that aims to utilize these properties to understand SAT problems in a deeper way [5–7, 26].

In this work we elevate the study of structure analysis of SAT to BFnS, thus

making the first step in structure analysis of BFnS. Our research questions are whether BFnS real-world instances have designated graph properties, distinguished from random BFnS instances, and whether such graph properties, if exist, affect the behavior of BFnS solvers. An affirmative answer to these questions can open the door to a more thorough study that seeks to improve BFnS tools performance by exploiting these graph properties.

To answer these questions, we chose graph representations of Boolean formulas that are natural for BFnS. Specifically, as BFnS is related to SAT, we examine the Variable Incidence Graph (VIG) and Clause to Variable Incidence Graph (CVIG) representations that are used for SAT structure analysis in [5–7]. In addition, we examine the Input Consensus Graph (ICG) representation that is used throughout the process of a BFnS solver called Back and Forth (BnF) [14]. The graph property that we choose to analyze in these graph representations, is graph modularity that is related to the graph community structure, and is used in SAT structure analysis. For benchmarks we use both random generated BFnS instances and so-called *real-world* or *industrial* BFnS instances, that originate from real problems and thus have some semantic interpretations.

We first show that unlike random instances, industrial instances admit very high graph modularity for the VIG representations. Typically, graph modularity ranges within the interval [0,1] and is defined high within [0.3,0.7], where for SAT benchmarks the average is 0.15 for random instances and just below 0.7 for industrial SAT instances [6]. For BFnS industrial instances the average modularity is 0.75, where for nearly all benchmarks is above 0.5 and for some it is as high as 0.9. The reason for that could be that in real world BFnS problems - some variables tend to be much more popular than others. These findings give a positive answer to our first research question and already serve as a promising start and encourage us to continue our exploration. While modularity distinctions were also noted for the other graph representations, we decided from this point to focus more on the popular VIG graph and its graph modularity (henceforth called VIG modularity).

Our next step is find how the VIG modularity of BFnS instances affects solvers performance. To meet this challenge, however, we need to find benchmarks with spanned VIG modularity over the [0,1] range. Note that we cannot use real-world benchmarks as we already showed that their VIG modularity is mostly focused in a small interval. As such, we construct a random BFnS instance generator called **ModQBF** with which we can control the VIG modularity of the generated instances. Our generator, based on a similar random SAT instance generator [20], can specifically produce BFnS

benchmarks of VIG modularity similar to those of industrial instances. Thus as a second use, **ModQBF** can produce random instances that are one-step closer to real world ones.

Using **ModQBF** , we deepen our analysis and check how various BFnS solvers perform on our randomly generated BFnS instances with controlled modularity. The solvers that we use are current state-of-the-art solvers: CADET [32], BnF [14], BFSS [1] and Manthan [22]. Our results give a positive answer to our second research question as we show that the average run-time of all solvers is indeed affected by the change of modularity. Specifically we show that as the VIG modularity increases, the average run-time of all solvers tends to decrease, whether moderately as in CADET and BFSS, or more drastically as BnF and Manthan.

Finally, we construct a monitoring method to better understand the relation between the structure of the BFnS benchmarks and the solvers performance. For that, we observe that all the solvers in mention work by iterative transformations and additions to the given formula as a part of the execution process. We white-boxed the solvers to find the precise places, called *iteration points*, in the solvers algorithms in which each change of the processed formula occurs. Then we extracted the structure of the formula in these iteration points for analysis. This gives us an abundance of data out of which we point out several observations of interest that relate the benchmarks VIG modularity and BFnS solvers performance.

Chapter 2

Background

2.1 Boolean Formulas and Boolean Functional Synthesis

Given a set of Boolean variables $\{x_1, ..., x_n\}$, a literal is an expression of the form x_i or $\neg x_i$. A clause c of size s is a disjunction of s literals, $l_1 \vee ... \vee l_s$. We denote $s = |c|$ to be the size of c. For a variable x and a clause c we denote $x \in c$, if c contains the literal x or $\neg x$. A CNF formula of length t is a conjunction of t clauses, $c_1 \wedge ... \wedge c_t$. We use the notation $c_j \in \varphi$ when a clause c_j is one of the clauses in the formula φ. A $k-CNF$ formula is a conjunction of k-sized clauses. A Boolean formula may be quantified, meaning that some or all of its variables are applied by either universal $\forall$ or existential $\exists$ quantifiers. A Boolean formula is *fully quantified* if all its variables are quantified. Given an unquantified Boolean formula φ, the problem in *Boolean Satisfiability (SAT)* is whether there is an assignment to the variables of φ under-which φ is evaluated to be *true*. An example of a SAT instance: $\exists \vec{y} : ((y_1 \vee \neg y_2 \vee y_5) \wedge (\neg y_2 \vee y_4 \vee y_5) \wedge (\neg y_1 \vee \neg y_3 \vee \neg y_4))$

Boolean Functional Synthesis

In Boolean functional synthesis (BFnS) we are given a specification φ in a form of a Boolean formula with two types of variables: input variables X_{in} and output variables X_{out}. The challenge is to find a function, called *skolem function*, $f : \{0,1\}^{X_{in}} \rightarrow \{0,1\}^{X_{out}}$ such that for every possible assignment $\vec{\sigma}$ to X_{in}, $\varphi(\vec{\sigma}, f(\vec{\sigma}))$ is evaluated to *true*. The output function can take various forms such as a Boolean formula [1] or a decision-list [14]. The related problem of *realizability* is whether such a function exists. Realizability is equivalent to the satisfiability of the fully quantified boolean formulas

with two alternations $\forall\exists$, that we call 2QBF. Indeed, a BFnS formula φ with $\vec{x}$ input variables and $\vec{y}$ output variables is realizable if and only if the formula $\forall\vec{x}\exists\vec{y}\varphi(\vec{x},\vec{y})$ is *true*. An example of a BFnS instance: $\forall\vec{x}\exists\vec{y}$: $((x_1 \vee \neg y_1 \vee y_2) \wedge (\neg x_1 \vee y_1 \vee \neg y_2) \wedge (x_2 \vee y_1 \vee y_2) \wedge (\neg x_1 \vee x_2 \vee \neg y_1 \vee \neg y_2))$

In this work we assume that the BFnS specification φ is given in CNF, and contains m clauses. We also assume that every clause in φ contains at least one output variable (otherwise φ is surely not realizable). When focusing on only the inputs of a clause, as is sometimes done in BFnS analysis (e.g. [14]), we define the *projection* c_{in} of a clause c on the input variables by removing all output variables from the clause c. Finally we sometimes refer to a BFnS instance with input variables X_{in}, output variables X_{out} and a set of clauses C, as a tuple (X_{in}, X_{out}, C). When we are not concerned about the type of each variable (i.e. whether input or output), we simply refer to a BFnS instance with X variables and C clauses as a pair (X, C).

2.2 Graphs

An undirected weighted graph G is a pair $G = (V, w)$ with a set of *vertices* V and a *weight function* $w : V \times V \to \mathbb{R}^{\geq 0}$. In this work w is symmetric (i.e. $w((v,u)) = w((u,v))$ for every $u, v \in V$) therefore we abuse notation and denote $w((u,v))$ by $w(u,v)$. A pair (u,v) of vertices is an *edge* in G if $w(u,v) > 0$. We denote $deg(v) = \sum_{u \in V} w(v,u)$ to be the (weighted) *degree* of a vertex v. A *bipartite weighted graph* $G_{bip} = (V_1 \cup V_2, w)$ is an undirected graph with two distinct sets V_1, V_2 of vertices, and a weight function w : $V_1 \times V_2 \to \mathbb{R}^{\geq 0}$ that maps between vertices in these distinct sets.

The *modularity* of a graph [16, 28, 29], is a measure of what is called the *community structure* in that graph. Intuitively, the vertices of a graph can be partitioned into sets called *communities*, where the size of every community is determined by how dense it is compared to the expected graph density in an Erdös Rényi random graph of with the same vertices and degrees [17]. The *community structure* (i.e. the partition of the graph to communities) then describes how random and uniform, or conversely - how distinctly partitioned, the graph is when looking at density of edges over the vertices. The modularity of a graph in a given partition is calculated as the ratio between weight of inner edges in the partition of vertices and total edges of the graph, compared to the expected ratio in a random Erdös Rényi graph on the same vertices and degrees. The overall modularity of a graph is then the maximal modularity over all possible graph partitions. Formally we have the

following.

Definition 1 (Graph Modularity) *Given a weighted undirected graph* $G = (V, w)$*, and a partition of its vertices* P*, the modularity of* G *with respect to* P *is defined to be:*

$$Q(G,P) = \sum_{P_i \in P} \left(\frac{\sum\limits_{u,v \in P_i} w(u,v)}{\sum\limits_{u,v \in V} w(u,v)} - \frac{\sum\limits_{v \in P_i} deg(v)^2}{\sum\limits_{v \in V} deg(v)} \right)$$

The modularity of a graph G *is then* $\max\{Q(G,P) | P \text{ is a partition of } V\}$.

2.3 BFnS Solvers

This work consists in part of analyzing the performance of BFnS solvers on BFnS instances and moreover white-boxing the solvers in order to deepen our analysis. For that, we give a description of four current state-of-the-art solvers that we use in this work.

CADET [32] is a BFnS solver that generalizes SAT-CDCL solvers to 2QBF, thus learns clauses implied by the given instance using a specified proof system until either a contradiction is reached or all variables have skolem functions and no further contradictions exist. CADET has quite a few extensions and we use the main one as in [32].

BnF [14] is a BFnS solver that decomposes the BFnS instance into input and output components, then constructs a graph structure called ICG (see Chapter 4.1) from the input component. Having that, BnF goes back and forth iteratively between finding cliques on the ICG and finding maximal satisfiable sets (MSS) on the output components, until no more MSS can be found. At every iteration step BnF produces another line in a growing decision list that eventually becomes the overall output solution.

BFSS [1] is a BFnS solver combined of two steps. In the first step that already solves most instances, BFSS uses verilog to find unate variables and other semantically implied dependencies to reduce the size of the formula. In the second step, BFSS uses a CEGAR approach [23] that iteratively finds counterexamples to candidate solutions, and refines them until no such counterexample is found.

Manthan [22] is a BFnS solver also combined of two steps. In the first step, as in BFSS, Manthan searches for unate variables to eliminate. Then however, it uses machine-learning and sampling techniques to learn candidate

solutions. In the second step, Manthan employs a refinement iterative procedure, different than CEGAR, that utilizes an unsatisifiability core found in a given error formula that contradicts the candidate solution, to produce an adjusted solution to offer for the next step and so on, until the error formula is not satisfiable, thus an overall solution is found.

Chapter 3

Previous Work

The problem of Boolean Functional Synthesis (BFnS) has its roots in the works of Boole, Lowenheim and Skolem [12, 25]. Since then it has been studied in various areas [24, 30, 33, 35] and there are many approaches towards its solution [1, 14, 22, 31, 32], some we detail in this work. BFnS was recently proven to be not only computationally hard to solve (specifically Π_2^P-hard), but also to have instances that admit super-polynomial solutions unless the Polynomial Hierarchy collapses [1]. One of the approaches to tame the computational-complexity of problems is the notion of decomposition and indeed there are works in BFnS that harness decomposition to construct better solvers [14, 34]. Specifically the work of [14] is inspired by exploration of sequential decomposition in computer science [18], and has a follow-up the area of reactive synthesis [4].

The world of BFnS has not yet had the time to delve deep into the understanding of some of the governing principles and dynamics behind BFnS solving or even real-world BFnS instances. In this work we study BFnS by deploying an analysis on structure of instances. We are inspired by a string of works in the field of SAT solving that analyzes SAT solvers performance through instance structure e.g. [5–7, 26]. Some of these works further refine the distinction between "real world" industrial SAT instances and randomly generated instances, created to produce instances at a required size and with predetermined features such as graph modularity [8, 9, 20, 21]. This string of works is a result of the attention that SAT received over the years which is at present day much greater than the attention given by research groups and researchers to BFnS.

Thus far, BFnS has been studied to try and better formulate instances, without understanding the underlying structures fully, for example in [14] and

[23]. While this approach has led to advancements and insights, by comparison to the world of SAT we can note that it avoids looking into what "real world" BFnS instances look like, and only employs factoring of the formula to different representations as a first step in solving them, not as a means of better understanding them.

This work seeks to be step towards evening out the playing field and allowing the world of BFnS to start "catching up to" the world of SAT in some respects. The BFnS random instance generator that we construct in this work (see Chapter 5) is an adaptation of some of these works to BFnS. We note that current BFnS generators are focused on generating instances that are either k-CNF adaptations to BFnS [15] or generating harder instances for specific tools [3]. As far as we know, no current BFnS generators control the modularity of BFnS instances. Next - we migrate the conclusions and analysis done in the world of SAT for graph modularity and give an analysis of the effects of modularity on BFnS solvers and instances, and we show the potential of this approach in improving the performance of solvers in the future, in ways both alike to those in the world of SAT and different from them.

This work builds on a trend in analyzing and understanding SAT that is still being researched and published even now, and much is yet to be discovered in that respect. Works such as [27] study features of SAT instances that present potential for analysis and appear congruent to modularity and community structure, and show that more is yet to be learned even in the field of SAT. The fact that analysing community structure is being researched in the SAT community for the last decade and more shows that it is an insightful tool for researchers, and this work sets out to put this tool in the hands of BFnS researchers for the first time.

Chapter 4

Analyzing Graph Structures of BFnS Instances

Our first objective is to find whether real-world BFnS benchmarks exhibit specific graph properties that are not found in random benchmarks. For that, we first provide several optional representations of BFnS formulas as graphs. We then focus on the graph modularity as the graph property that we analyze on each of the graph representations. We describe this in Chapter 4.1. Then, in Chapter 4.2 we perform our analysis and report our results.

4.1 BFnS Graph Representations and Properties

We first describe the graphs that we use to represent the Boolean formulas as. Although these can be found in previous works (e.g. [6, 14]), we elaborate here to keep the paper self-contained. We explore three different graph representations that are natural to explore in the BFnS setting: the Variable Incidence Graph (VIG) and Clause to Variable Incidence Graph (CVIG) that are used for analyzing SAT instances [5–7], and a version of the Input Consensus Graph (ICG), that is used in the BnF solver [14]. We will use as an example the formula given before: $\forall\vec{x}\exists\vec{y}$: $((x_1 \vee y_1 \vee y_2) \wedge (\neg x_1 \vee \neg y_2 \vee y_3) \wedge (x_2 \vee y_3 \vee y_4) \wedge (\neg x_1 \vee x_2 \vee \neg y_1 \vee \neg y_4))$

Definition 2 *The Variable Incidence Graph (VIG) of a BFnS instance φ with variables X and clauses C, is a graph $G_\varphi^{vig} = (X, w^{vig})$ where*

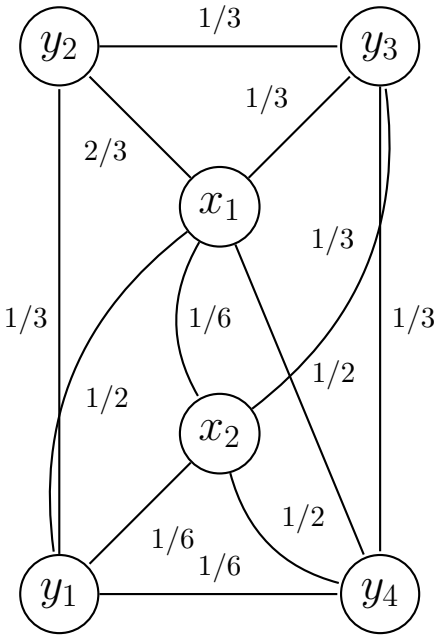

Figure :4.1 VIG of the example BFnS instance $(Q = 0.163194)$

$$w^{vig}(x_i, x_j) = \sum_{c \in C \ s.t. \ x_i, x_j \in c} \frac{1}{\binom{|c|}{2}}$$

The VIG represents the relationships between every two variables and the amount of clauses they both appear in. As discussed in [5–7], this graph may allow insights about the overall structure of the instance and the centrality of certain variables in it. The VIG is the main representation that we analyze in our work. The VIG of the running example is given in figure 4.1.

Definition 3 *The Clause to Variable Incidence Graph (CVIG) of a BFnS instance φ with variables X and clauses C is a bipartite graph $G_{\varphi}^{cvig} = (X \cup C, w^{cvig})$ where*

$$w^{cvig}(x, c) = \begin{cases} \frac{1}{|c|} & \text{if } x \in c \\ 0 & \text{otherwise} \end{cases}$$

The $CVIG$ is a bipartite graph that describes a relation between "variable-vertices" and "clause-vertices", in which the degrees of every "variable-vertex" represents the amount of clauses the corresponding variable appears in. Again,

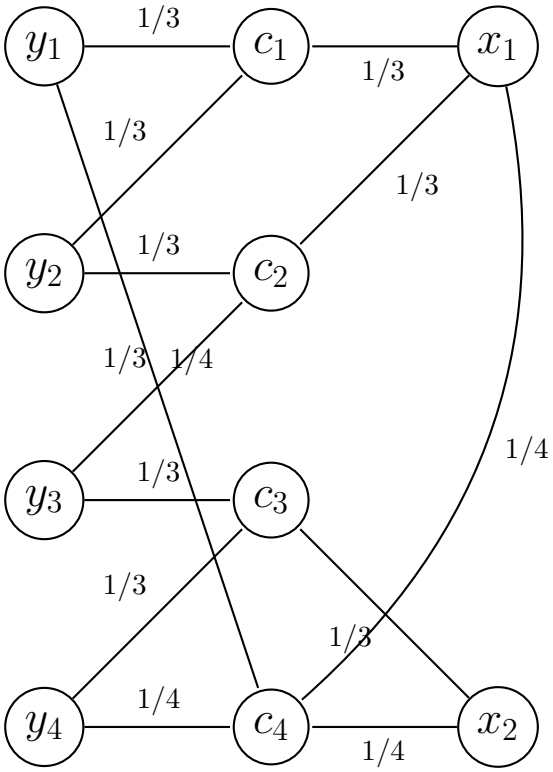

Figure :4.2 CVIG of the example BFnS instance $(Q = 0.28559)$

as discussed in [5–7], the CVIG may allow us to better understand the centrality of certain variables and the amount of clauses a variable can appear in. The CVIG of the running example is given in figure 4.2.

Definition 4 *The Input Consensus Graph (ICG) of a BFnS instance φ with clauses C, is a graph $G^{icg}_{\varphi} = (V^{icg}, w^{icg})$ where $V^{icg} = \{c_{in} | c \in C\}$ and*

$$w^{icg}(c_i, c_j) = \begin{cases} 1 & \text{if } c_i \wedge c_j = True \\ 0 & \text{otherwise} \end{cases}$$

The ICG represents the relationships between the input components of different clauses. Note that $c_i \wedge c_j = True$ for clauses c_i, c_j in which no variable appears in one clause with its negation in the other. The ICG of the running example is given in figure 4.3.

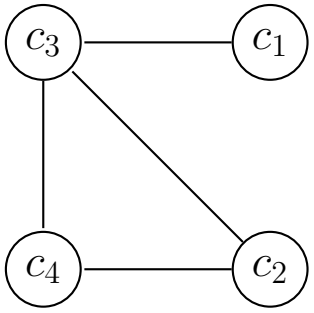

Figure :4.3 ICG of the example BFnS instance ($Q = 0.333333$)

Using graph modularity

In this work we focus on *graph modularity* (see Chapter 2) as the property to analyze for the various graph representations. This is inspired by previous works on SAT analysis as [5, 6]. We also note that generation of random graphs with a specified modularity is more manageable, as we show in Chapter 5. Other graph properties such as the fractal dimension of a graph [5], are left for future work.

Since finding the graph modularity of a given graph is an NP-hard problem [13], we use the Louvain approximation, as is commonly used in [6, 11], which gives the best-to-date lower bound on the modularity of a graph. Then, similarly to the analysis done for SAT, we take an approximation Q of the VIG modularity and test whether $Q \in [0.3, 0.7]$. If Q is within these values, the graph is said to have *high modularity*. High modularity points out a certain partition of the graphs vertices into *communities* wrt to which most edges are internal. This interval is set since lower values tell of no partition of the graph, and higher values are very rare [16, 28, 29]. For bipartite graphs we use the definition given by [10], which is not required for the rest of this work. We note, however, that CVIG and VIG modularities tend to be similar.

4.2 BFnS Structure Analysis

We next analyze the graph modularity of BFnS benchmarks for the graph representations that we defined. for that, we construct the three graph representations from each BFnS benchmark in our repository, and calculate the graphs (approximated) modularity. Since our purpose is to explore features of instances that come from ”real world” problems, we choose benchmarks

that are used to analyze and compare BFnS solvers. Specifically we use benchmarks from the 2QBF track of the 2016-2018 QBFEval, an annual competition for solvers of QBF problems that includes BFnS. The timeout that we used for modularity calculation is 24hr, see Chapter 8 for machine setup. Modularity calculation for several benchmarks timed-out, thus is missing from our analysis. To finalize the distinction between industrial and random benchmarks, we also analyze random BFnS instances, generated by a 2QBF random generator called $ChenInterian$ [15].

We report our analysis in Table 4.1. It is apparent that instances from QBFEval admit very high modularity for their VIG and CVIG of average 0.75 and 0.82 respectively, that is above the [0.3,0.7] high modularity definition. Moreover, apart from *qshifter* and Q_2_3 which is a random generator (it is practically $ChenInterian$), all benchmarks have modularity above 0.5. The average modularity for QBFEval instances is even higher than the average found for SAT instances which is 0.7 [6]. On the other hand the random generator $ChenInterian$ produces instances with VIG modularity typically around the value of 0.15 or less. Note also that QBFEval instances tend to admit very low modularity for their ICG, much below the 0.15 expected in a randomly generated graph.

All in all, our analysis confirms our first research question and shows that industrial BFnS benchmarks indeed admit high VIG and CVIG modularity, specifically compared to random generated instances. We henceforth choose to focus on the VIG graph that is more popular, and refer to the modularity of VIG as *VIG modularity*.

Input/Output projection analysis

Since BFnS is a problem that relates input and output variables, we also ran some analyses on graphs that consider only the input or only the output variables (i.e. the input projections and output projections of the BFnS formulas) and tested them for their VIG and CVIG modularity. The differences that we found between input and output projections were not significant, although we did find that VIG modularity tends to be slightly higher in the input projection. As such, we decided to leave the projection analysis for future work.

Table :4.1 Average initial modularity in QBFEval

Benchmark Family	Number of Benchmarks	Average Initial Modularity value		
		VIG	CVIG	ICG
total	637	0.7579	0.8232	0.0153
sketch	11	0.7217	0.8277	0.0079
mutexP	7	0.7850	0.8468	0.0109
Sorting_networks	42	0.8071	0.8592	0.0004
ltl2dpa	1	0.6944	0.8207	0.0012
terminator	69	0.7957	0.8580	0.0074
qshifter	6	$<$ 0.0001	0.1241	0.9590
toy	5	0.7271	0.8327	0.0219
ltl2dba	4	0.6770	0.7922	0.0014
Reduction-finding	99	0.7482	0.7689	0.0083
Model_instances	6	0.5112	0.7280	0.0063
RankingFunctions	87	0.8659	0.9128	0.0009
hardwareFixpoint	96	0.7919	0.8680	0.0040
irqlkeapclte	65	0.8839	0.9162	0.0032
Q_2_3	20	0.0738	0.2675	0.0059
arith	11	0.9024	0.9281	0.0008
driver	6	0.7189	0.8239	0.1058
disjunctive_decomposition	9	0.6722	0.7685	0.0028
hwmcc	2	0.8507	0.8939	0.0015
ltl2aig-comp	4	0.6794	0.7933	0.0009
cycle_sched	6	0.7040	0.8095	0.0133
wmiforward	40	0.8317	0.8735	0.0100
Selection-hard	20	0.5155	0.7376	0.0018
mult_matrix	6	0.6783	0.8313	0.0086
tree	5	0.6348	0.7308	0.0227
amba	5	0.6337	0.6762	0.0187
genbuf	5	0.6270	0.6815	0.0122

Chapter 5

Generating BFnS Instances with Controlled Modularity

Our next objective is to construct a generator that produces random BFnS instances with a desired modularity. Such a generator allows us to continue our exploration and to answer our second research question of how BFnS solvers run-time is affected by the changes of the benchmarks modularity. More than that, since we established that BFnS industrial instances tend to have high VIG modularity, such a generator can produce random instances that are one-step closer to real-world benchmarks. Our construction follows the works done in the SAT community in which similar generators for SAT are discussed [9, 20, 21].

5.1 Generator Algorithm Description

Based on the *Community Attachment* algorithm for random SAT generator described in [20], we provide an algorithm **ModQBF** for the generation of random BFnS k-CNF instances exhibiting desired modularity, see Algorithm 1.

The input for **ModQBF** is n, m, p that are respectively number of variables, number of clauses, and ratio of input-variables to total variables. In addition we have k as the number of variables per clause, c as the desired number of communities and Q as the desired modularity of the instance. The output of **ModQBF** is a BFnS k-CNF instance with n variables, out of which $\lfloor p * n \rfloor$ are input variables, m clauses, c communities, and modularity of at least Q.

Since BFnS differentiates between input and output variables, we make the

following considerations for using **ModQBF** when assigning input/output variables. We first require that each clause contains at least one output variable, as otherwise the instance is trivially nonrealizable (i.e. has no solution). Note however that **ModQBF** can still generate nonrealizable instances. Next, we added an option to require every clause to contain at least one input variable, so as to differentiate the problem from SAT. Finally note that the parameter p in practice also controls the ratio of input to output variables.

As an overview, **ModQBF** works as follows. First, the algorithm partitions the variables into c different communities, and each of these communities into respective parts of input and output variables. Each community has the same number of variables, as in [20], and is assigned the same number (plus or minus one) of input variables. Next, two types of clauses are generated: clauses that contain variables from only one community, or clauses that contain variables from k distinct communities. The choice between clause types is controlled by a probability, calculated by following the strategy described in [20], that promises VIG modularity bounded above given Q. Our main addition compared to Community Attachment is that in each clause an output variable is selected, and optionally an input variable is selected as well. This change is required to handle two problems not found in the case of SAT: First, nonrealizability may be trivial if no output variable is found in any clause, and second, if input variables occur in only a few clauses, then the instances degenerate in practice to become too similar to SAT instances.

In detail, **ModQBF** is described as follows. *Rand* is a random generator function that we overload to simplify notation. Hence $Rand([i, j])$ returns a uniformly distributed real value in the interval $[i, j]$, and $Rand(\mathcal{S})$ returns a uniformly distributed random element from the discrete set S. In lines 1-2 of Algorithm 1, the returned BFnS instance I is initialized, and the variable d, that indicates the number of variables per community, is defined. In lines 3-6 the communities are defined, first input variables of each community, then output variables of each community, then the union of both as whole community. In lines 7-27 there is a loop that generates new clauses (and assures there are no repeated variables in any clause). In lines 9-18 communities are chosen for each variable in the new clause - lines 9-12 are for the case that all variables are from the same community, and lines 13-18 are for the case that each variable is in a distinct community. Lines 19-26 assign variables to the clauses, from the community assigned to each variable, Line 19 makes sure an output variable is chosen and line 20 **is optional** and makes sure an input variable is chosen.

Algorithm 1 ModQBF

Require: Integers n, m, k, c, Real values $Q, p \in (0, 1)$, s.t.:
$3 \leq k \leq min(c, \frac{n}{c})$ and $c \leq \lfloor p * n \rfloor \leq n - c$
Output: A k-CNF BFnS instance with n variables m clauses, and modularity $\geq$ Q

:1 $\mathcal{I} = \emptyset$
:2 $d = \frac{n}{c}$
:3 **for** $i = 1$ **to** c **do**
:4 $Com_i^{in} = \{\lceil (i-1)d \rceil + 1, \ldots, \lceil (i-1)d \rceil + \lceil pd \rceil\}$
:5 $Com_i^{out} = \{\lceil (i-1)d \rceil + \lceil pd \rceil + 1, \ldots, \lceil id \rceil\}$
:6 $Com_i = Com_i^{in} \cup Com_i^{out}$
:7 **end for**
:8 **while** $|\mathcal{I}| < m$ **do**
:9 $Cl =$ **false**
:10 **if** $Rand([0, 1]) \leq Q + \frac{1}{c}$ **then** {same community}
:11 $j = Rand(\{1, \ldots, c\})$
:12 **for** $i = 1$ **to** k **do**
:13 $C[i] = j$
:14 **end for**
:15 **else** {distinct communities}
:16 **for** $i = 1$ **to** k **do**
:17 **repeat**
:18 $j = Rand(\{1, \ldots, c\})$
:19 **until** $j \notin C[0, ..., i-1]$
:20 $C[i] = j$
:21 **end for**
:22 **end if**
:23 $Cl = Cl \vee Rand(\{-1, 1\}) \cdot Rand(Com_{C[1]}^{out})$
:24 $Cl = Cl \vee Rand(\{-1, 1\}) \cdot Rand(Com_{C[2]}^{in})$
:25 **for** $i = 3$ **to** k **do**
:26 **repeat**
:27 $v = Rand(Com_{C[i]})$
:28 **until** $v \notin Cl$
:29 $Cl = Cl \vee Rand(\{-1, 1\}) \cdot v$
:30 **end for**
:31 $\mathcal{I} = \mathcal{I} \cup Cl$
:32 **end while**
:33 **return** $\mathcal{I}$

The choice between two types of clauses according to a certain probability is done to control the modularity of the resulting instance [20]. Each variable is chosen randomly and uniformly among its community wrt to its distinction (input or output). Moreover communities for each variable in a clause are chosen uniformly and randomly, with the requirement being that there are either k distinct communities or one repeated community. To get an instance with modularity Q, probability of $P = Q + \frac{1}{c}$ is used, as discussed in [20]. Specifically, note that if the distinction between input and output variables was removed, **ModQBF** would yield the same instances that Community Attachment in [20] does. As a result, all arguments regarding modularity from [20] apply to our model, and specifically Theorem 1.

Theorem 1 *Let φ be an instance generated by **ModQBF** with modularity value Q. Then the modularity of G_{φ}^{vig} is at least Q.*

Chapter 6

Generated Instance Characteristics

In this chapter we give a comparison of BFnS instances generated using **ModQBF** and *ChenInterian* presented in [15], to test whether or not prevalent generators are able to produce industrial instances easily, as well as how our generator fares when trying to produce them. First, we describe the parameters of **ModQBF** and *ChenInterian*, renaming parameters to fit our description. We generated 10 instances of each of the possible combinations of following the values.

- $n \in \{100, 200, 300\}$
- $m \in \{3n, 3.5n, 4n, 4.5n, 5n\}$
- $p \in \{0.2, 0.4, 0.6, 0.8\}$
- $k \in \{3, 5\}$
- For **ModQBF** only: $Q \in \{0.05i | 1 \leq i \leq 19\}$
- For **ModQBF** only: $c \in \{n/20, n/10\}$
- For *ChenInterian* only: $a \in \{i \in \mathbb{N} | 1 \leq i \leq k-1\}$

It is apparent that there are many more **ModQBF** instances than there are *ChenInterian* instances, but the analysis is split between the two to make the number of instances irrelevant. We aim to study both the inherent features of the two instance generators, for use of industrial benchmarks for solvers.

Table :6.1 Average initial values of features in benchmarks generated by *ChenInterian*

Benchmarks Grouped By	Number of Benchmarks	Average Initial Modularity Value		
		VIG	CVIG	ICG
total	3600	0.1205	0.3489	0.0089
p=0.2	900	0.1152	0.3474	0.0170
p=0.4	900	0.1262	0.3506	0.0086
p=0.6	900	0.1263	0.3507	0.0057
p=0.8	900	0.1144	0.3470	0.0043
k=3	1200	0.1769	0.4471	0.0054
k=5	2400	0.0923	0.2998	0.0106
m/n=3.0	720	0.1417	0.3691	0.0090
m/n=3.5	720	0.1285	0.3563	0.0090
m/n=4.0	720	0.1181	0.3468	0.0089
m/n=4.5	720	0.1105	0.3396	0.0088
m/n=5.0	720	0.1039	0.3329	0.0088
a=1	1200	0.1328	0.3727	0.0032
a=2	1200	0.1360	0.3736	0.0076
a=3	600	0.0959	0.3014	0.0131
a=4	600	0.0896	0.2995	0.0185

We give a table for each of the generated instance groups. Table 6.2 shows average initial feature values for **ModQBF** generated instances, and Table 6.1 shows average initial feature values for *ChenInterian* generated instances.

A quick glance at the tables reveals a rather interesting fact: while **ModQBF** could be given a configuration to produce "real-world"-like features in a rather small instance - *ChenInterian* seems to be much harder to manipulate to get different values of modularity in the VIG and CVIG of the produced instance. From looking at other families of benchmarks in QBFLIB [1] it is clear that by changing the parameters one can get different modularity values, but due to the nature of the generator *ChenInterian* expected modularity values are very hard to control.

[1] to access the different families of benchmarks in QBFLIB, go to `http://www.qbflib.org/benchmarks_start_page_old.php`

Table :6.2 Average initial values of features in benchmarks generated by $MODQBF$

Benchmarks Grouped By	Number of Benchmarks	Average Initial Modularity Value		
		VIG	CVIG	ICG
total	45600	0.5080	0.6196	0.0063
p=0.2	11400	0.5084	0.6203	0.0086
p=0.4	11400	0.5077	0.6190	0.0059
p=0.6	11400	0.5080	0.6192	0.0054
p=0.8	11400	0.5080	0.6199	0.0051
k=3	22800	0.5155	0.6515	0.0047
k=5	22800	0.5006	0.5877	0.0078
m/n=3.0	9120	0.5126	0.6248	0.0063
m/n=3.5	9120	0.5099	0.6216	0.0063
m/n=4.0	9120	0.5076	0.6191	0.0063
m/n=4.5	9120	0.5062	0.6172	0.0062
m/n=5.0	9120	0.5039	0.6152	0.0062
c=5	7600	0.4934	0.5857	0.0105
c=10	15200	0.5080	0.6175	0.0076
c=15	7600	0.5110	0.6274	0.0035
c=20	7600	0.5134	0.6325	0.0050
c=30	7600	0.5145	0.6370	0.0033
Q=0.05	2400	0.1466	0.3768	0.0066
Q=0.10	2400	0.1597	0.3830	0.0066
Q=0.15	2400	0.1846	0.3957	0.0066
Q=0.20	2400	0.2186	0.4146	0.0066
Q=0.25	2400	0.2592	0.4395	0.0066
Q=0.30	2400	0.3049	0.4695	0.0065
Q=0.35	2400	0.3526	0.5020	0.0065
Q=0.40	2400	0.4024	0.5376	0.0065
Q=0.45	2400	0.4511	0.5729	0.0064
Q=0.50	2400	0.5008	0.6094	0.0063
Q=0.55	2400	0.5496	0.6451	0.0063
Q=0.60	2400	0.5987	0.6811	0.0062
Q=0.65	2400	0.6493	0.7183	0.0061
Q=0.70	2400	0.6996	0.7552	0.0061
Q=0.75	2400	0.7490	0.7914	0.0060
Q=0.80	2400	0.7993	0.8283	0.0059
Q=0.85	2400	0.8405	0.8584	0.0058
Q=0.90	2400	0.8819	0.8887	0.0057
Q=0.95	2400	0.9042	0.9049	0.0057

Chapter 7

Monitoring Solver Behavior on BFnS Structures

We so far answered our two research questions: we showed that VIG modularity of BFnS industrial benchmarks is high, and that solver performance is affected by the instances modularity. In this chapter we monitor the on-going change in VIG modularity of a given BFnS instance while being processed through the BFnS solvers. By doing that we get an abundance of data of how the solvers work, out of which we already make several observations that further relate the solvers performance to VIG modularity.

7.1 Finding Solver Iteration Points

To monitor the solvers work process, we make use of an observation that during execution of every BFnS solver that we studied, the original instance keeps changing through the solving process. For example, for CADET that is CDCL-based, learned clauses are added to the original formula. Hence, as the formula is changing, so does its structure.

Therefore we first identify in each BFnS solver designated points, called *iteration points*, in which the structure of the instance being solved is changing. We then monitor the solvers at these points and update the instance and the graph representations at each of these point. For the purpose of this research no solver behaviors are altered, but rather the data is written to a log through the run of the solvers.

For CADET and BnF, identifying iteration points is more or less straightforward. First see that since CDCL is an iterative process, in which every

iteration produces an additional learned clause to the instance, there is an iteration point for CADET whenever a new clause is learned. For BnF note that since every new line in the output decision list can be thought of as a clause being learned, there is an iteration point for every line in the decision list that is gradually written.

For BFSS and Manthan identifying iteration points task is more challenging. BFSS has three distinct types of iteration points: The first is the propagate-unates iteration point in which a discovered unate variable is removed from all clauses, and entire clauses may be removed from the original formula. The second is the propagate-dependencies iteration point in which a variable is replaced by a simple function, followed by removing clauses that are now trivially true. Finally in the learning phase, there is an iteration point whenever a new counterexample is learned as a clause, much like CADET iterations. Manthan, on the other hand, has two types of iterations points. The first type is propagate-unates iteration point as in BFSS. The second type of iteration point is where an unsatisfiability core of a counterexample is learned. We note that since the unsatisfiability core is given in DNF, the structure analysis in practice is harder and takes longer to run for Manthan. This explains why some of our analyses for Manthan timed-out (as shown in Chapter 9).

7.2 Analysis via Monitoring Iteration Points

By using the iteration points that we defined, we ran all four BFnS solvers through **ModQBF** benchmarks with controlled modularity, in order to observe how VIG modularity changes for the various graph structures, as the benchmarks being processed by the solvers. The amount of data that we have gathered that way is enormous and can take a while to fully comprehend and process. For example, a single solved instance on 300 variables can have more than 300 iterations when running through CADET, where each iteration requires computing graph representations and features analysis. Nevertheless, we present below several observations that we can already make, that further describe the connection between the instances and their modularity, and also shed more lights on how these solvers work. More details can be found in Chapter 9.

We first compare the number of iterations that every tool makes on non-trivial instances against its initial VIG modularity. This is given in Figure 8.1:d. We see that as initial VIG modularity increases, CADET and BnF require more iterations, while Manthan requires less. BFSS seems ag-

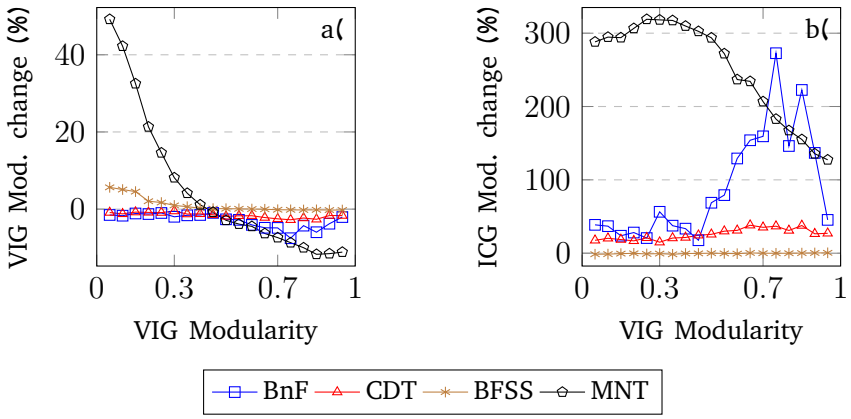

Figure :7.1 Modularity change against VIG modularity for **ModQBF** instances: :7.1a - average VIG modularity change, :7.1b - average ICG modularity change

nostic to instance VIG modularity in the amount of iterations it requires. Comparing with Figure 8.1:c that describes the tools run-time against the initial VIG modularity, we observe that although requiring more iterations to solve higher VIG modularity instances - both BnF and CADET solve these instances much faster. For example 300 BnF iterations for Q=0.9 take roughly 10 times less to execute than 20 BnF iterations for Q=0.5. This finding demonstrates an additional connection between tools run-time and VIG modularity.

Next, we check how VIG modularity changes (in percentage) from before to after the instance is solved, compared to the initial VIG modularity. This is given in Figure 7.1:a. As the x axis describes the initial VIG modularity, The y axis presents the change in modularity in percentage: percentages above 0 indicate an increase in modularity and percentages below 0 indicate a decrease. Note that while VIG modularity does not change much for three of the tools, Manthan's solving process affects VIG modularity decidedly, going in a smooth curve from increasing it for low VIG modularity instances to decreasing VIG modularity for high VIG modularity instances. It is interesting to see that of all tools, Manthan that relies on ML techniques, demonstrates such a clean curve.

Finally, we run a similar analysis and this time check how the ICG modularity is changed (in percentage) by the solvers as initial VIG modularity increases. As Figure 7.1:b shows, the ICG modularity almost always increases through solving, even by roughly 300 percent for BnF and Manthan at various VIG modularity values. Manthan again has the most pronounced effect. These findings, that connect VIG modularity to ICG modularity are surprising,

since apart from BnF, no other tool does any use with the ICG structure.

Chapter 8

Solvers Performance on Specified-Modularity Instances

Using our generator **ModQBF** , we can now answer our second question of interest, of how does the VIG modularity of BFnS instances affects performance of BFnS solvers. For that, we use **ModQBF** to generate instances with various values of VIG modularity, solve each instance using each solver, and compare results to see whether or not VIG modularity affects the solvers run-time. Note that using QBFEval instances to answer this question may not do, since these mostly admit modularity of around 0.75. The solvers that we explore are CADET, BnF, BFSS and Manthan, see details in Chapter 2.3.

Machine setup All tests and algorithms were executed on HPC clusters at The Open University of Israel, comprised of 44 compute node machines, each node having 2 Intel Xeon Gold 6130 16 Cores or Xeon Gold 6230 20 Cores CPU processors, 192 GB RAM. The run-time for the solvers was limited to 8 hours until timeout, and each running job was allocated 4GB of RAM. Solver analysis was allotted additional 8 hours.

Analysis setup We analyze the BFnS solver performance over instances with various VIG modularity generated by **ModQBF** . We generated 45600 instances, varying between 3-CNF and 5-CNF and ranging in size of 100-300 variables. The ratio of clauses/variables is ranged from 3 to 5 to situate it around the phase transition for 3-CNF. We also used the option of having an input variable in every clause. The VIG modularity is generated from Q=0.05 to Q=0.95 in 0.05 steps.

We ran our analysis to see whether VIG modularity affects BFnS solvers performance. Our results are described in Figure 8.1, see Table 9.3 in Chapter 9 for full details.

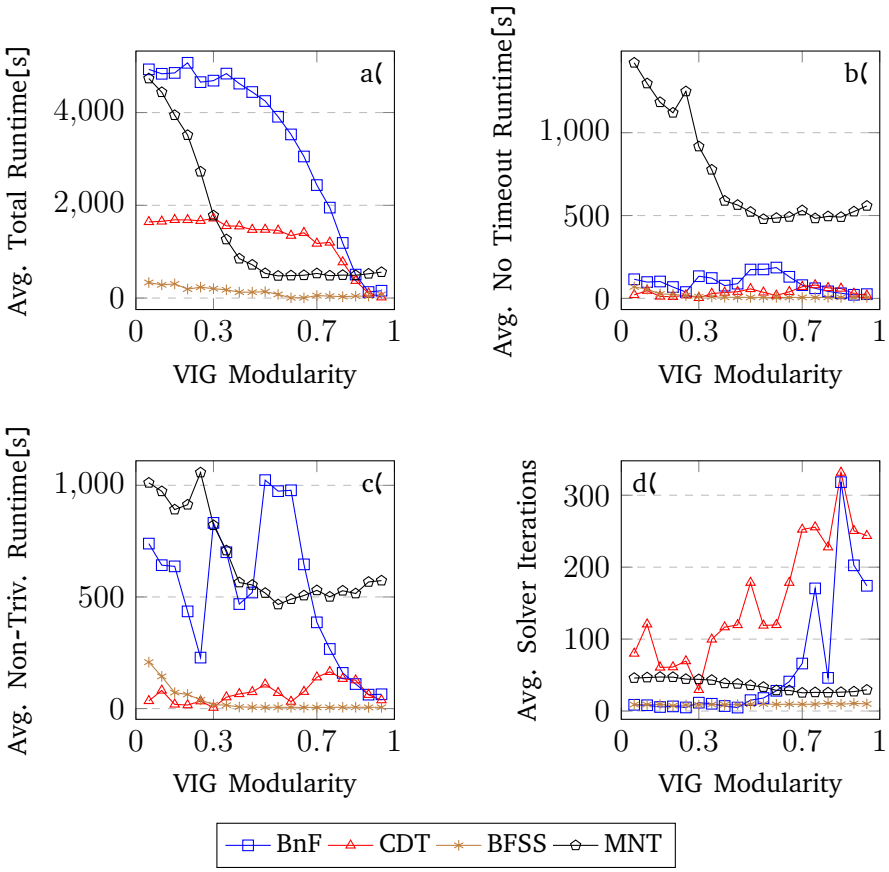

Figure :8.1 Time analyses against VIG modularity for **ModQBF** instances: :8.1a - average total run-time, :8.1b - average non-timeout run-time, :8.1c - average non-trivial instance run-time, :8.1d - average number of iterations (discussed in Chapter 7.2(

Figure 8.1:a (see caption at upper-right corner of each figure) depicts the average solver run-time (in seconds) against initial VIG modularity. In this figure we included timed-out instances that counted as 8 hours. It is clear from this figure that the average run-time for all solvers decreases as VIG modularity goes up, specifically after Q=0.3 which is the lowest bar for high-modularity as defined in [29]. BnF seems to be affected the most by VIG modularity, only starting to reduce in run-time around a modularity of Q=0.5. Manthan run-times decrease more sharply starting at Q=0.3, and CADET is shown to be affected by VIG modularity as well, decreasing run-times considerably starting at Q=0.75, which is the average VIG modularity of QBFEval instances. Note that in all this, BFSS curve is extremely low, as also depicted in additional figures. We believe that the reason for that is that BFSS solves

most instances that we generated rapidly fast, using no more than a few first-step iterations.

Since timed-out instances can affect the average run-time considerably (although curves still imply that less timeout occurs as modularity go up, see Chapter 9 for details), we wanted to check the curves filtering these out. Figure 8.1:b depicts the average solver run-time (in seconds) against initial VIG modularity. In this figure we filtered out timed-out instances. Manthan maintains the descent into a plateau, but it is clear that it times out much more on lower VIG modularity instances. BnF and CADET are much much closer to BFSS in average run-time for solved instances, which shows that all 3 solvers are very fast for instances they manage to solve, unlike in the case of Manthan. The curve for BFSS is again almost flat.

The two analyses given above already demonstrate a connection between instance VIG modularity and solver performance. Nevertheless, since many instances terminate almost immediately, we can learn more about the solvers behavior by observing *non-trivial instances* - instances that did not time-out but required solvers to perform at least a single iteration (as defined in Chapter 7). For that, Figure 8.1:c depicts solver run-times of non-trivial instances, against VIG modularity. The time scale is at most 1050 seconds since all instances that did not meet that time have timed-out and were disregarded. Also here we see that average run-time is affected by modularity, for example for BnF that drops dramatically starting from Q=0.6 which is more or less the average modularity for QBFEval instances, or for Manthan that drops in Q=0.3 then slowly stabilizes. With respect to Manthan it is interesting to observe that its average run-time is increased when timeouts and immediate instances are disregarded. We suspect that the reason is that Manthan may immediately find correct solutions in instances with no unate variables, resulting in no iterations being made. More generally, our experiments suggest that Manthan may reduce its run-time, and still maintain its advantage in solving more benchmarks, by adapting the entire first stage of BFSS (not only the unates step).

Chapter 9

Generator Test Results In Depth

9.1 Analyses Completion

To complete the the analyses done in Chapters 8 and 7, we provide Figure 9.1 below.

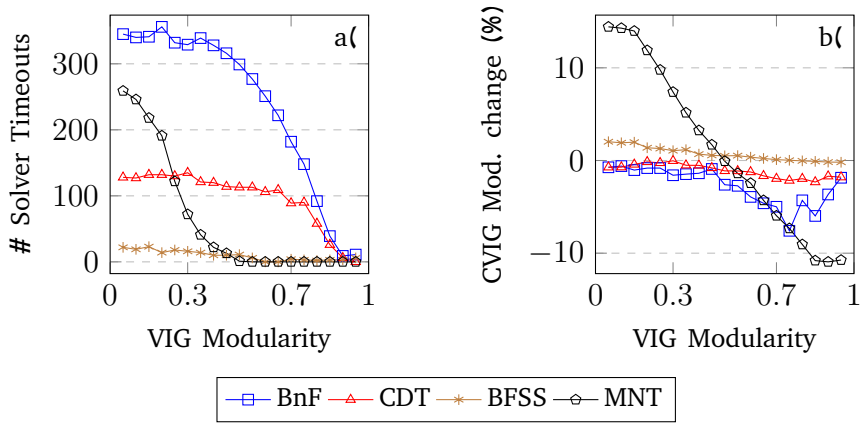

Figure :9.1 A few more analyses with respect to VIG modularity on **ModQBF** instances :9.1a - number of solver timeouts, :9.1b - average CVIG modularity change

In Figure 9.1:a the number of solver timeouts is plotted against **ModQBF** VIG modularity, for completion of Figures 8.1:a-c. BnF and CADET can be seen to struggle with instances of VIG modularity $Q \leq 0.7$, and timeouts only decrease dramatically for higher modularity. Manthan struggles with instances of modularity $Q \leq 0.3$, but then quickly flattens out and is able to solve many more instances. The curve for BFSS is again rather flat, and it may be the case that Manthan seems to be a somewhat more exaggerated

version of it is due to the exclusion of dependency analysis and the differences between the second stages of both solvers.

In Figure 9.1:b the average change of CVIG modularity is plotted against **ModQBF** VIG modularity, for completion of Figures 7.1:a-b. Comparison with Figure 7.1:a shows that solver behaviour for CVIG modularity is indeed very much in keeping with what was observed for VIG modularity. It is interesting to note that changes inflicted on CVIG modularity seem to be a lesser magnitude version of the changes inflicted to VIG modularity.

9.2 Further Data and Observations

We give Tables 9.1, 9.2, and 9.3 that describe the full detail of the analysis for solvers run on BFnS instances generated by **ModQBF** . Each table is described in its caption. In these tables the data relates to instances that did not time out and for which the relevant solver performed at least one iteration. The flag DNS is used if a solver was not successful in solving or starting the iterative process, on any benchmark in the family. Moreover, for reasons of brevity - CDT and MNT will respectively represent CADET and Manthan. Double horizontal separators separate different dissections of the data (by Q value, k value, etc.).

Table 9.1 shows analysis of performance on non-trivial instances for different solvers, detailing both number of non-trivial instances for each solver, and average number of iterations made. Note that Manthan is the most prolific solver in terms of instances it requires iterations to solve, leading the next runner up in CADET by more than a 50% margin. BnF seems to be highly affected by VIG modularity again, as it requires iterations for more instances as VIG modularity increases. CADET and BFSS are well rounded out as far as VIG modularity is concerned, requiring iterations for roughly the same number of instances at each modularity. Manthan requires increasingly more iterations as modularity increases to $Q = 0.3$ and then less iterations as modularity increases past that point. It is interesting to note that BFSS is the only solver to require iterations for more 3-CNF instances than 5-CNF instances, and all other solvers show the opposite behaviour.

Table 9.2 shows analysis of change of modularity for all 3 graphs studied in this work (VIG, CVIG, ICG) on non-trivial instances for different solvers. It has been analyzed wrt to VIG modularity above, and so we only give supplemental observations here. Note that all solvers tend to behave very differently for 3-CNF and 5-CNF instances, with the exception being Man-

than for ICG modularity.

Table 9.3 shows analysis of run-times for 3 different dissections of instances, that was analyzed in Chapter 8. BnF and CADET seem to work faster as the ratio of input variables to total variables, p, increases. BFSS seems to work slower as p increases. All solvers aside for Manthan seem to work faster for 3-CNF than they do for 5-CNF, while Manthan works faster for 5-CNF than for 3-CNF.

Table :9.1 Average performance in solving process by different solvers for **ModQBF** instances

Grouping Based on	# Non-Trivial Instances				Avg. # Iterations			
	BnF	CDT	MNT	BFSS	BnF	CDT	MNT	BFSS
total	7511	21494	34863	13039	91.67	150.35	36.59	9.25
$p = 0.2$	4884	5269	8603	5541	23.15	432.99	42.43	11.41
$p = 0.4$	1764	5855	8960	4152	325.13	160.43	40.41	8.20
$p = 0.6$	808	4270	8892	2293	2.27	1.14	34.70	7.20
$p = 0.8$	55	6100	8408	1053	1.20	1.00	28.52	6.46
$k = 3$	121	3392	12066	11998	1.70	1.00	34.45	9.78
$k = 5$	7390	18102	22797	1041	93.14	178.34	37.72	3.15
$m/n = 3.0$	2006	4105	8905	4099	125.08	82.22	39.47	9.67
$m/n = 3.5$	1753	4213	8217	3420	79.07	213.05	43.26	9.91
$m/n = 4.0$	1606	4131	6570	2936	103.38	205.47	30.78	13.33
$m/n = 4.5$	1223	4320	5698	1537	26.04	128.01	32.97	4.11
$m/n = 5.0$	923	4725	5473	1047	109.55	125.89	32.60	1.52
$c = 5$	1789	4366	5794	1924	43.84	81.70	26.44	7.39
$c = 10$	2840	7572	11828	4125	79.44	117.26	31.10	8.94
$c = 15$	812	3194	5736	2384	114.13	245.22	53.88	10.28
$c = 20$	1135	3305	5928	2214	85.02	194.97	33.52	9.88
$c = 30$	935	3057	5577	2392	208.85	183.03	44.23	9.65
$Q = 0.05$	261	1248	1914	657	8.68	79.98	45.82	8.91
$Q = 0.10$	256	1214	1950	626	8.04	120.62	46.55	8.99
$Q = 0.15$	267	1225	1963	638	5.60	60.65	47.28	9.24
$Q = 0.20$	255	1174	1991	684	6.41	61.06	47.08	7.35
$Q = 0.25$	272	1172	2035	650	4.93	69.45	44.38	8.33
$Q = 0.30$	275	1120	2102	645	11.59	29.50	44.02	8.39
$Q = 0.35$	295	1174	2091	678	10.10	99.59	43.08	9.12
$Q = 0.40$	277	1130	2070	674	6.94	116.64	38.45	8.82
$Q = 0.45$	294	1143	1991	679	4.79	119.68	37.76	8.93
$Q = 0.50$	302	1118	1945	722	15.09	178.49	35.78	9.41
$Q = 0.55$	327	1091	1891	706	18.07	118.97	33.31	9.80
$Q = 0.60$	356	1141	1819	760	28.11	119.78	28.69	9.26
$Q = 0.65$	388	1106	1721	703	40.94	178.49	28.61	9.40
$Q = 0.70$	409	1086	1703	761	66.03	252.40	25.32	9.32
$Q = 0.75$	464	1070	1643	737	170.49	255.63	25.94	9.44
$Q = 0.80$	556	1063	1587	705	46.10	227.78	25.59	10.89
$Q = 0.85$	607	1077	1500	699	318.21	331.54	26.21	9.45
$Q = 0.90$	762	1071	1481	645	202.53	250.54	26.98	10.53
$Q = 0.95$	888	1071	1466	670	174.12	243.65	29.44	9.96

Table :9.2 Average changes in modularity over solving process by different solvers for **ModQBF** instances

Grouping Based on	Avg. Change of Modularity Through Solving (percents)											
	VIG				CVIG				ICG			
	BnF	CDT	MNT	BFSS	BnF	CDT	MNT	BFSS	BnF	CDT	MNT	BFSS
total	3.78-	1.84-	1.95-	0.35	3.51-	1.29-	0.69-	0.52	99.69	25.49	259.33	0.39-
$p = 0.2$	1.12-	3.38-	3.04-	0.70	1.07-	2.11-	2.45-	0.80	23.72	32.66	161.39	0.75-
$p = 0.4$	10.89-	3.33-	3.42-	0.17	10.53-	2.50-	2.42-	0.37	383.46	58.84	291.15	0.32-
$p = 0.6$	0.23-	0.07-	1.25-	0.02	0.22-	0.13-	0.06	0.23	0.64	0.68	333.56	0.27
$p = 0.8$	0.04-	0.05-	0.01-	0.02	0.03-	0.13-	2.15	0.24	2.13	0.50	294.13	1.84
$k = 3$	0.18-	0.08-	3.59-	0.38	0.13-	0.15-	4.00-	0.55	1.35-	0.47	289.98	0.49-
$k = 5$	3.87-	2.14-	1.17-	0.06-	3.59-	1.51-	1.11	0.03	101.21	29.19	249.57	0.46
$m/n = 3.0$	5.79-	1.34-	2.77-	0.01-	5.18-	1.02-	2.07-	0.07	159.90	19.58	356.91	0.21
$m/n = 3.5$	3.98-	2.49-	2.27-	0.14	3.59-	1.76-	1.62-	0.36	100.25	35.66	308.23	0.85-
$m/n = 4.0$	2.55-	2.43-	1.35-	1.41	2.32-	1.69-	0.33	1.67	54.71	26.52	206.17	1.31-
$m/n = 4.5$	2.33-	1.74-	1.02-	0.21	2.29-	1.22-	0.81	0.34	59.87	23.59	190.68	0.26
$m/n = 5.0$	3.53-	1.24-	1.68-	0.04-	3.59-	0.82-	0.36	0.02-	100.21	22.10	181.09	0.23
$c = 5$	3.70-	1.93-	6.71-	0.06-	3.48-	1.57-	3.14-	0.39	71.89	27.54	191.48	0.58-
$c = 10$	4.00-	1.58-	4.38-	0.39	3.70-	1.13-	2.33-	0.62	96.35	21.62	226.49	0.44-
$c = 15$	2.91-	1.46-	1.45-	0.33	2.74-	1.09-	0.74-	0.40	145.45	31.70	419.69	0.53-
$c = 20$	3.73-	2.70-	1.42	0.60	3.44-	1.68-	1.24	0.71	130.83	32.65	295.28	0.15-
$c = 30$	4.14-	1.86-	3.96	0.37	3.78-	1.15-	3.07	0.37	189.33	20.45	408.00	0.06
$Q = 0.05$	1.52-	0.89-	49.20	5.64	0.73-	0.73-	14.42	2.03	38.65	17.55	288.12	1.42-
$Q = 0.10$	1.70-	1.23-	42.23	5.07	0.60-	0.72-	14.28	1.91	36.84	20.30	294.65	1.27-
$Q = 0.15$	1.13-	0.78-	32.52	4.54	1.03-	0.42-	13.97	1.98	23.77	18.58	293.90	0.61-
$Q = 0.20$	1.28-	0.85-	21.32	2.05	0.82-	0.15-	11.89	1.38	28.14	16.82	306.55	0.18-
$Q = 0.25$	1.00-	0.90-	14.59	1.70	0.82-	0.26-	9.78	1.29	20.61	20.92	318.96	1.05-
$Q = 0.30$	1.99-	0.51-	8.14	0.96	1.59-	0.03-	7.39	1.06	56.42	14.96	318.15	0.58-
$Q = 0.35$	1.61-	1.28-	4.12	0.63	1.47-	0.51-	5.16	1.17	37.52	20.65	317.36	1.52-
$Q = 0.40$	1.52-	1.23-	1.13	0.24	1.38-	0.53-	3.25	0.69	33.45	21.41	309.76	0.43-
$Q = 0.45$	0.98-	1.47-	0.78-	0.13	0.91-	0.80-	1.70	0.55	17.66	24.43	302.54	0.31-
$Q = 0.50$	2.64-	1.88-	2.90-	0.11	2.61-	1.12-	0.07-	0.51	68.78	25.70	293.70	0.01-
$Q = 0.55$	2.79-	1.73-	3.88-	0.08	2.73-	1.13-	1.39-	0.53	79.19	30.00	272.10	0.29-
$Q = 0.60$	4.09-	1.72-	4.47-	0.02	3.93-	1.24-	2.44-	0.36	129.15	31.10	236.80	0.67-
$Q = 0.65$	4.84-	2.18-	6.32-	0.04-	4.61-	1.66-	4.30-	0.22	154.00	38.10	234.32	0.33
$Q = 0.70$	4.87-	2.52-	7.39-	0.16-	5.00-	1.96-	5.96-	0.09	159.35	35.27	206.73	0.03-
$Q = 0.75$	7.54-	2.76-	8.70-	0.19-	7.58-	2.16-	7.33-	0.03	272.69	36.80	182.90	0.14-
$Q = 0.80$	4.28-	2.31-	10.00-	0.22-	4.30-	1.96-	9.06-	0.04-	146.17	30.96	167.29	0.08-
$Q = 0.85$	5.99-	2.63-	11.71-	0.19-	5.98-	2.32-	10.80-	0.09-	222.58	37.69	155.07	0.12
$Q = 0.90$	3.78-	1.67-	11.54-	0.24-	3.67-	1.70-	10.94-	0.17-	136.74	26.45	135.60	0.30
$Q = 0.95$	2.06-	1.67-	11.14-	0.23-	1.86-	1.80-	10.74-	0.19-	45.41	27.10	127.34	0.50

Table :9.3 Average runtime in solving process by different solvers for **ModQBF** instances

Grouping Based on	Average Time Taken on Non Timeout Instances [s]				Average Time Taken on Non-Trivial Instances [s]				Average Total Time Taken on Instances [s]			
	BnF	CDT	MNT	BFSS	BnF	CDT	MNT	BFSS	BnF	CDT	MNT	BFSS
total	89.65	36.43	720.79	14.78	424.24	68.82	679.70	31.62	3276.58	1252.04	1469.95	131.40
$p = 0.2$	117.43	160.86	704.34	6.62	227.27	236.73	678.28	6.38	2605.62	5509.46	1137.27	6.62
$p = 0.4$	491.60	19.24	738.10	24.71	1156.64	35.17	696.34	54.47	13610.13	158.76	1758.54	24.71
$p = 0.6$	5.39	2.30	750.31	21.31	44.54	2.59	691.29	62.87	96.61	2.30	1786.17	46.57
$p = 0.8$	2.66	2.32	692.21	6.22	2.49	2.45	651.17	6.27	2.66	2.32	1209.47	452.51
$k = 3$	2.53	2.31	885.91	22.35	4.59	3.50	881.90	33.67	2.53	2.31	2412.16	22.35
$k = 5$	234.77	77.16	572.76	7.08	431.11	81.06	572.69	7.99	7368.25	2612.05	576.48	241.33
$m/n = 3.0$	71.48	20.77	714.68	9.64	270.69	43.69	710.46	8.87	2141.61	122.11	714.68	41.25
$m/n = 3.5$	83.47	40.90	844.49	6.41	355.39	82.65	873.85	6.31	2489.31	698.91	1305.64	9.56
$m/n = 4.0$	58.88	51.58	591.18	45.15	265.96	100.49	557.73	114.97	2845.19	1526.28	2552.93	54.61
$m/n = 4.5$	103.33	25.49	689.77	6.94	572.32	44.22	486.20	8.00	4055.57	1961.75	1586.70	455.83
$m/n = 5.0$	140.32	44.68	750.13	5.45	967.87	73.15	686.05	4.30	5024.82	2044.43	1257.40	100.48
$c = 5$	165.22	11.90	438.28	6.70	609.83	17.44	381.62	7.21	1958.90	68.83	438.28	6.70
$c = 10$	107.56	28.81	600.97	8.04	464.59	51.60	571.83	8.78	2845.30	870.43	727.69	13.73
$c = 15$	22.61	47.96	992.67	40.30	136.36	95.98	869.75	105.72	4691.95	2383.63	3436.66	462.68
$c = 20$	58.68	60.52	786.18	5.50	297.48	124.72	790.98	5.15	3563.31	1464.57	964.23	5.50
$c = 30$	58.65	44.64	979.24	20.92	350.42	96.06	904.41	41.30	3904.84	1970.28	2750.89	292.31
$Q = 0.05$	115.29	20.99	1421.92	69.25	738.97	34.38	1011.16	208.04	4930.97	1642.34	4733.89	335.05
$Q = 0.10$	98.38	47.00	1297.08	55.11	642.39	81.33	973.12	144.24	4835.54	1653.52	4438.09	284.49
$Q = 0.15$	101.84	11.80	1183.82	26.61	637.01	18.81	891.33	71.64	4854.67	1687.31	3942.91	305.02
$Q = 0.20$	68.45	9.78	1120.20	24.33	435.37	15.92	913.10	63.15	5072.57	1685.40	3513.52	193.17
$Q = 0.25$	38.18	19.18	1249.09	16.76	228.30	33.26	1057.19	38.85	4655.65	1667.42	2724.60	234.26
$Q = 0.30$	133.79	3.44	916.69	9.86	831.83	4.54	821.31	17.84	4687.72	1719.80	1779.06	203.08
$Q = 0.35$	122.69	28.82	776.24	8.99	700.26	51.14	708.03	15.15	4839.63	1556.38	1263.30	177.93
$Q = 0.40$	76.89	35.40	588.64	6.61	468.13	65.62	564.70	7.05	4623.79	1549.33	849.63	127.09
$Q = 0.45$	88.88	40.34	563.51	6.22	519.79	74.58	553.69	6.58	4442.39	1474.55	717.29	126.70
$Q = 0.50$	174.11	57.13	521.86	6.05	1023.32	108.57	517.52	5.70	4247.95	1477.31	533.64	138.63
$Q = 0.55$	174.83	36.59	478.05	5.83	973.87	70.62	466.21	5.77	3909.72	1457.78	478.05	77.99
$Q = 0.60$	185.93	17.47	485.02	5.60	977.73	31.22	490.06	5.72	3528.01	1347.44	485.02	5.60
$Q = 0.65$	130.25	39.56	491.74	5.57	645.96	75.57	506.31	5.28	3052.51	1407.91	491.74	5.57
$Q = 0.70$	79.73	70.22	531.09	5.87	386.28	140.47	529.48	5.31	2436.40	1176.65	531.09	53.94
$Q = 0.75$	61.15	80.55	482.86	5.85	266.94	164.69	500.54	5.46	1949.85	1199.48	482.86	41.88
$Q = 0.80$	42.27	63.33	494.77	6.04	159.83	133.75	527.87	5.89	1188.59	775.00	494.77	30.06
$Q = 0.85$	30.73	59.45	490.59	6.12	109.93	127.07	515.51	6.08	505.96	374.22	490.59	42.16
$Q = 0.90$	21.73	28.98	523.85	5.72	62.38	61.87	567.16	5.56	130.05	101.08	523.85	41.76
$Q = 0.95$	25.47	18.08	558.28	5.83	64.09	38.03	573.17	5.74	157.96	18.08	558.28	78.00

Chapter 10

Conclusion

In this work we studied structure analysis for the BFnS problem. We first showed that unlike random instances, industrial BFnS instances admit very high VIG modularity. This motivated us to provide methods that can allow us to better understand VIG modularity for BFnS. One of the methods is **ModQBF** , a BFnS instance generator that provides random instances with a desired VIG modularity. Using **ModQBF** we showed that VIG modularity affects the performance of various state-of-the-art BFnS solvers. Specifically, the run-time for some of these solvers decreases dramatically as modularity increases. Another method for structure analysis that we showed, is how to white-box the solvers to mark designated iteration points in the solvers process in order to monitor the ongoing change in the instance structure through the BFnS solving process. Using that, we were able to show various behaviors of interest, that relate tools performance to VIG modularity.

A point to consider is how well can **ModQBF** simulate industrial instances. While **ModQBF** can be adjusted to generate instances with VIG modularity set to $Q = 0.75$, same as the average VIG modularity of industrial instances, we believe that there may be more properties that industrial instances may posses that we did not yet capture. As evidence, we found that the execution of BFSS on **ModQBF** almost always never reached step 2 of its process. This is while for industrial benchmarks BFSS reached step 2 for roughly 17 percent of the cases. Obviously more research is required in this case.

Finally, as this work is inspired by modularity analysis for SAT, there is the question of how much the current impact of modularity analysis in SAT solvers, reflects on future deployment of structure analysis in BFnS solvers. One must remember however, that the BFnS research is far behind SAT, which enjoys much more attention and has current solving tools that are suf-

ficiently good for many real-world problems. Due to this distinction, analyses that have proven to improve SAT solvers, even if minutely, may yet prove to be beneficial for the problem of BFnS. In that aspect, this work is the first step towards potentially utilizing structure analysis in the area of BFnS.

A point should be made that the use of modularity and community structure in the analysis of SAT has conflicting evidence to its relevance. However, as the tool of community structure has been studied this past decade in various ways by various groups and all concluded in publish-worthy papers describing new insights on SAT, this work has its place in the world of BFnS as the pioneering venture into the field of analysis of BFnS instances as an insightful tool. Not all insights are yet pointed out and not all potential is yet realized, but future works building upon this work will tackle all those questions and more.

At this current point in time this work presents more questions than it answers, and the case could be made that many more questions should be posed.Questions like: is there any semantic interpretation to the different representations of instances? What does high modularity mean for BFnS instances? What does it mean for a benchmark to increase its modularity when going through a certain solver? are all questions we do not have many more answers for than insights and gut feelings this work already suggested. As in other areas in computer science, we may find ways to exploit the various properties of BFnS instances graph structures long before interpreting their meaning to the full extent.

Time will tell if the field of BFnS will benefit from mimicking trends from the field of SAT. If this work is any reference - there is reason for optimism that it will, and that is good news, due to the uneven amount of attention between both fields. This work will continue, because the raw potential in using the world of SAT as a compass to navigate the world of BFnS is yet to be realized, and will require more studies that will begin to answer some of the questions posed above.

Bibliography

[1] Akshay, S., Chakraborty, S., Goel, S., Kulal, S., and Shah, S. What's hard about boolean functional synthesis? In *Computer Aided Verification - 30th International Conference, CAV, Proceedings, Part I* (2018).

[2] Akshay, S., Chakraborty, S., John, A. K., and Shah, S. Towards parallel boolean functional synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS, Proceedings, Part I* (2017).

[3] Amendola, G., Ricca, F., and Truszczynski, M. Random models of very hard 2QBF and disjunctive programs: An overview. In *19th Italian Conference on Theoretical Computer Science* (2018).

[4] Amram, G., Bansal, S., Fried, D., Tabajara, L. M., Vardi, M. Y., and Weiss, G. Adapting behaviors via reactive synthesis. In *Computer Aided Verification - 33rd International Conference, CAV, Proceedings, Part I* (2021).

[5] Ans□otegui, C., Bonet, M. L., Gir□aldez-Cru, J., and Levy, J. The fractal dimension of SAT formulas. In *Automated Reasoning - 7th International Joint Conference, IJCAR. Proceedings* (2014).

[6] Ans□otegui, C., Bonet, M. L., Gir□aldez-Cru, J., Levy, J., and Simon, L. Community structure in industrial SAT instances. *J. Artif. Intell. Res. 66* (2019), 443–472.

[7] Ans□otegui, C., Bonet, M. L., and Levy, J. On the structure of industrial SAT instances. In *Principles and Practice of Constraint Programming - CP, 15th International Conference, Proceedings* (2009).

[8] Ans□otegui, C., Bonet, M. L., and Levy, J. Towards industrial-like random SAT instances. In *IJCAI, Proceedings of the 21st International Joint Conference on Artificial Intelligence* (2009).

[9] Barak-Pelleg, D., Berend, D., and Saunders, J. C. A model of random industrial SAT. *CoRR abs/1908.00089* (2019).

[10] Barber, M. Modularity and community detection in bipartite networks. *Physical review. E, Statistical, Nonlinear, and Soft Matter Physics 76*, 6 Pt 2 (2007), 066102–066102.

[11] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment 2008* (2008).

[12] Boole, G. *The Mathematical Analysis of Logic*. Philosophical Library, 1847.

[13] Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering* (2008).

[14] Chakraborty, S., Fried, D., Tabajara, L. M., and Vardi, M. Y. Functional synthesis via input-output separation. In *2018 Formal Methods in Computer Aided Design, FMCAD* (2018).

[15] Chen, H., and Interian, Y. A model for generating random quantified boolean formulas. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence* (2005).

[16] Clauset, A., Newman, M. E. J., and Moore, C. Finding community structure in very large networks. *Physical Review E* (2004).

[17] Erd□os, P., and R□enyi, A. On random graphs I. *Publicationes Mathematicae Debrecen* (1959).

[18] Fried, D., Legay, A., Ouaknine, J., and Vardi, M. Y. Sequential relational decomposition. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS* (2018).

[19] Fried, D., Tabajara, L. M., and Vardi, M. Y. Bdd-based boolean functional synthesis. In *Computer Aided Verification - 28th International Conference, CAV, Proceedings, Part II* (2016).

[20] Gir□aldez-Cru, J., and Levy, J. A modularity-based random SAT instances generator. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI* (2015).

[21] Gir□aldez-Cru, J., and Levy, J. Locality in random SAT instances. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI* (2017).

[22] Golia, P., Roy, S., and Meel, K. S. Manthan: A data-driven approach for boolean function synthesis. In *Computer Aided Verification - 32nd International Conference, CAV, Proceedings, Part II* (2020).

[23] John, A. K., Shah, S., Chakraborty, S., Trivedi, A., and Akshay, S. Skolem functions for factored formulas. In *Formal Methods in Computer-Aided Design, FMCAD* (2015).

[24] Kukula, J. H., and Shiple, T. R. Building circuits from relations. In *Computer Aided Verification, 12th International Conference, CAV, Proceedings* (2000).

[25] L., L. über die auflösung von gleichungen im logischen gebietekalkul. *Mathematische Annalen 69* (1910).

[26] Li, C., Chung, J., Mukherjee, S., Vinyals, M., Fleming, N., Kolokolova, A., Mu, A., and Ganesh, V. On the hierarchical community structure of practical boolean formulas. In *International Conference on Theory and Applications of Satisfiability Testing* (2021), pp. 359–376.

[27] Mull, N., Fremont, D. J., and Seshia, S. A. On the hardness of SAT with community structure. *CoRR abs/1602.08620* (2016).

[28] Newman, M. E. J. Fast algorithm for detecting community structure in networks. *Physical Review E* (2004).

[29] Newman, M. E. J., and Girvan, M. Finding and evaluating community structure in networks. *Physical Review E* (2004).

[30] Rabe, M. N., and Seshia, S. A. Incremental determinization. In *Theory and Applications of Satisfiability Testing - SAT - 19th International Conference, Proceedings* (2016).

[31] Rabe, M. N., and Tentrup, L. CAQE: A certifying QBF solver. In *Formal Methods in Computer-Aided Design, FMCAD* (2015).

[32] Rabe, M. N., Tentrup, L., Rasmussen, C., and Seshia, S. A. Understanding and extending incremental determinization for 2QBF. In *Computer Aided Verification - 30th International Conference, CAV, Proceedings, Part II* (2018).

[33] Solar-Lezama, A. Program sketching. *Int. J. Softw. Tools Technol. Transf. 15*, 5-6 (2013), 475–495.

[34] Tabajara, L. M., and Vardi, M. Y. Factored boolean functional synthesis. In *2017 Formal Methods in Computer Aided Design, FMCAD* (2017).

[35] Zhu, S., Tabajara, L. M., Li, J., Pu, G., and Vardi, M. Y. Symbolic LTLf synthesis. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI* (2017).

תוכן העניינים

תקציר

בעיית הסינתזה הבוליאנית היא בעית היצירה של פונקציות בוליאניות ממפרט בוליאני המתאר קשרים בין משתני קלט ופלט. מאחר ולבעיה זו שימושים רבים, כמו למשל בעיצוב מעגלים מודפסים, פתרון נוסחאות בוליאניות מכומתות לעייפה וכן סינתזה ריאקטיבית - נעשים כל העת מאמצים להבין ולחקור את בעיית הסינתזה הבוליאנית.
בעבודה זו, ישנו מחקר המעמיק את ההבנה של בעיית הסינתזה הבוליאנית על ידי ניתוח של מבני גרפים אופיניים למופעים של בעיית הסינתזה הבוליאנית. עבודה זו מונעת מרצף עבודות בעולם של בעיית הספיקות הבוליאנית, SAT, בהן תכונות של גרפים מקבילים, כמו למשל מודולריות של גרפים, שימשו כלי מחקרי על מנת להבין את הביצועים של כלי פתרון לבעיית SAT על מופעים תעשייתיים של הבעיה, המגיעים מהעולם האמיתי.
ראשית, נראה כי בשונה ממופעים אקראיים של בעיית הסינתזה הבוליאנית, מופעים תעשייתיים או מופעים "מהעולם האמיתי" מציגים מודולריות גרף גבוהה עבור הגרפים המייצגים המנותחים בעבודה זו, ואף בצורה משמעותית יותר מכפי שנמצא במקרה של בעיית SAT ברצף העבודות המוזכר לעיל. לאור כך, נציג מחולל אקראי של מופעי סינתזה בוליאנית, אשר מאפשר חילול מופע עם מודולריות מבוקשת, ונשתמש בו על מנת לחקור את ההשפעה של מודולריות גרף מייצג עבור מופע של בעיית הסינתזה הבוליאנית על ביצועיהם של כלי פתרון מודרניים עבור הבעיה. לבסוף, נציץ "מתחת למכסה המנוע" ונביט אל תוך המימוש הפנימי של אותם כלי פתרון על מנת לקבוע באיזו צורה הם מושפעים ומשפיעים על מודולריות של מופע לאורך תהליך הפתרון.
הממצאים של מחקר זה מעידים כי מודולריות של מופע משפיעה באופן ישיר על הביצועים של כלי פתרון והתנהגותם על מופע.

האוניברסיטה הפתוחה
המחלקה למתמטיקה ולמדעי המחשב

ניתוח מבני בבעיית הסינתזה הבוליאנית

עבודת תזה זו הוגשה כחלק מהדרישות לקבלת תואר
"מוסמך למדעים" .M.Sc במדעי המחשב
באוניברסיטה הפתוחה
המחלקה למתמטיקה ומדעי המחשב

על-ידי
זיו אבישר

בהנחיית **ד"ר דרור פריד**

יוני 2023