BINARY ENCODINGS FOR SOLVING AD-HOC CONSTRAINTS

WANG RUIWEI

A THESIS SUBMITTED FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

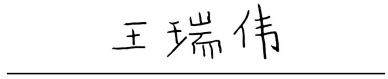
2022

Advisor:	Examiners:
Prof. Roland Yap	Prof. Joxan Jaffar
	Prof. Kuldeep S. Meel

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Wang Ruiwei
June 2022

Acknowledgments

I acknowledge the financial support from the National University of Singapore. I would like to thank my advisor, Prof. Roland Yap, for all of the guidance and advice I have received during my thesis. I am thankful for the opportunities he provided for me to grow professionally. I thank my thesis examiners, Prof. Joxan Jaffar and Prof. Kuldeep S. Meel, for their insightful comments which helped me improve my thesis. I am also very grateful to, Prof. Yuanlin Zhang and Dr. Wei Xia, for all the helpful discussions and collaborations.

I thank the professors who have offered me lectures or shared inspiring suggestions with me during my PhD studies. I thank my senior colleagues: Dr. Suhendry Effendy, Dr. María Andreína Francisco Rodríguez and Dr. Thanh-Toan Nguyen, for their generous help. I thank my peer PhD candidates: Yuyi Zhong, Yahui Song, Hewen Wang, and Xiao Liang Yu, for all the cheerful conversations and collaborations.

Most importantly, I thank my parents, sisters and brothers for their unconditional love and continuous encouragement.

Abstract

Historically, work on Constraint Satisfaction Problems (CSPs) began with binary CSPs and algorithms proposed to enforce Arc Consistency (AC) on binary constraints. In addition, several well-known binary encoding methods can be used to transform non-binary constraints into binary constraints. However, in more recent times (from the 1990s), research has focused on non-binary constraints and Generalized Arc Consistency (GAC) algorithms. Existing results and "folklore" suggest that AC algorithms with binary encodings do not compete with GAC algorithms on the original non-binary constraints. In this thesis, we propose new binary encodings to efficiently solve non-binary ad-hoc constraints, such as the table, Ordered Multi-valued Decision Diagram (MDD) and regular constraints.

We give a specialized algorithm called HTAC to enforce AC on constraints encoded by Hidden Variable Encoding (HVE). Preliminary experiments show that our AC algorithm on HVE encoded constraints is competitive to state-of-the-art GAC algorithms on the original non-binary constraints and faster in some instances. This result is surprising and is contrary to the "folklore" on AC versus GAC algorithms.

We use a new binary encoding called Bipartite Encoding (BE) to transform table constraints into binary constraints. AC on the BE encoded constraints can achieve a higher level of consistency than GAC on the original constraints. In addition, we give a specialized algorithm to enforce AC on the BE encoded constraints. Experiments show that BE with our AC algorithm can significantly outperform the state-of-the-art table GAC algorithms.

We propose a binary encoding called Direct Tree Binary Encoding (DTBE) to transform the MDD and regular constraints into Binary Constraint Trees (BCTs), where a BCT is a set of binary constraints with a tree structure. We show that BCT constraints can be exponentially smaller than the corresponding MDD and regular constraints. Furthermore, we introduce reduction rules to simplify BCT constraints, and giving a specialized algorithm to enforce AC on the BCT constraints. Our experiments show that the BCT AC algorithm can be significantly faster than MDD GAC algorithms.

In addition to AC algorithms, we also tailor the CNF encodings of binary constraints to transform BCT constraints into Conjunctive Normal Form (CNF), making them suitable for SAT solvers. We compare the propagation strength of the BCT CNF encodings and experimentally evaluate the encodings on a range of benchmarks. Experimental results show that the CNF encodings of BCT constraints can outperform those of MDD constraints on various benchmarks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

The Constraint Satisfaction Problem (CSP) with finite domain variables is a well-known NP-Complete problem [SC06], where the famous 3-SAT problem can be regarded as a special case of CSPs modelled with clause constraints over Boolean variables. A wide range of combinatorial problems in real-life can be modelled as CSPs, e.g., the product configuration problem can be modelled as CSPs with constraints such that the components of a product are modelled as variables and each solution of the CSPs satisfying the constraints over the product components corresponds to a feasible instance of the product [VFA99]. Additionally, many kinds of constraints have been proposed to model different problems. For example, the AllDifferent constraint [Rég94] can be used to model the air traffic flow management [BB04], the sequence constraint [BC94a] and global cardinality constraint [Rég96] can be used to model the car sequence problem [RP97], and the regular constraint [Pes04] can be used to model the nurse rostering problem.

The constraints used for modelling CSPs can be categorized as special purpose global constraints [BC94a, BCR10] and ad-hoc constraints [CY06b] (also called generic constraints). The special purpose global constraints impose specific semantics on their constraint relations, therefore, they can have efficient specialized solving algorithms which exploit the constraint semantics. However, this is also a weakness, i.e., there exists constraint relations which cannot be easily modelled with well-studied global constraints. Unlike the special purpose global constraints, the ad-hoc constraints are more expressive and can have arbitrary constraint relations over finite domain variables. They are very useful for modelling the specific constraints which are difficult to model with special purpose global constraints, e.g., the still life problems can be modelled with different ad-hoc constraints [CY06a].

We will focus on the ad-hoc constraints (generic constraints) which can be defined with different representations, such as tables [BR97, Lec11, WXYL16, DHL+16], non-ordinary tables [KW07, GHLR14, JN13, MDL15, VLDS17, ALM20], decision diagrams [CY10, PR14, VLS18], automatons [Pes04, QW06] and context-free grammars [QW06, Sel06]. Many real-life constraints can be modelled with the ad-hoc constraints. For example, the Nonogram

puzzles[1] can be modelled with the regular constraints and Ordered Multi-valued Decision Diagram (MDD) constraints [GSS11], the transition constraints used for modelling planning problems can be represented as short table constraints [GNNS15], and various specific constraints used for modelling the related-key differential characteristic problems can be encoded as table constraints [GMS16, GLMS20].

Typically, CSPs are solved by backtrack search with Generalized Arc Consistency (GAC) propagators, where the GAC propagators are used to reduce the search space. This is especially so given that there are different GAC propagators for different constraint types [Rég11a]. For example, we can respectively use Arc Consistency (AC) propagators and Unit Propagation algorithms to enforce GAC on the binary constraints and clause constraints [Bes06]. A number of algorithms (GAC propagators) have been proposed to enforce GAC on various ad-hoc constraints exploiting their constraint representations, such as many table GAC algorithms [BR97, GJMN07, LS06, Ull07, Lec11, LLY12, MVHD12, XY13, WXYL16, DHL+16] and MDD GAC algorithms [CY10, GSS11, PR14, VP18, VLS18].

In addition to design efficient GAC propagators for non-binary constraints (whose constraint arity is at least 3), we can also use binary encoding methods to transform non-binary constraints into binary constraints, such as the dual encoding [DP89], hidden variable encoding [RPD90] and double encoding [SW99b], and then using efficient AC propagators to enforce AC on the binary constraints. Compared with non-binary constraints, binary constraints have much simpler formats, making it easier to explore their properties and develop sophisticated AC algorithms [Bes99].

The study of CSPs began with binary CSPs and AC propagators on binary constraints. Before the 1990s, a large body of algorithms were proposed to solve binary constraints [Kum92]. In addition, many GAC algorithms for non-binary constraints are also extended from AC algorithms, e.g., the AC3 [Mac77a], AC4 [MH86], AC7 [BFR95] and AC2001/3.1 [BR01, ZY01] algorithms are generalized to the GAC3 [Mac77b], GAC4 [MM88], GAC-schema [BR97] and GAC2001/3.1 [BRYZ05] algorithms. From a theoretical perspective, finite domain binary CSPs are already NP-complete. All non-binary constraints can be solved by encoding into binary constraints. However, many existing algorithms of binary constraints cannot work very well on the binary encodings of non-binary constraints. For example, the experimental results given in [SW99a, Lec11] show that the GAC propagators of the AllDifferent constraints and table constraints can be significantly faster than AC propagators with binary encodings. Along with efficient GAC algorithms come, non-binary constraints have become more and more important, which has led to a "folklore" suggesting that it is better to solve non-binary constraints with GAC propagators rather than encoding them into binary constraints and using AC propagators.

[1] https://webpbn.com/survey/

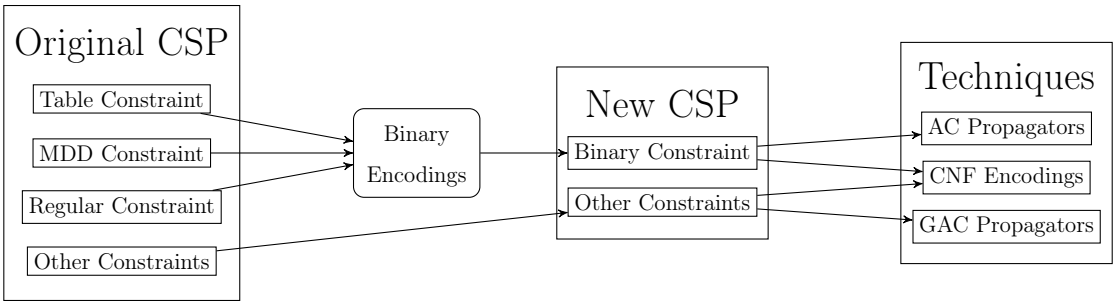


Figure 1.1: Improve the GAC propagators and CNF encodings of ad-hoc constraints from a binary encoding perspective.

Another important technique [Pre09] used to solve CSP instances is encoding constraints into Conjunctive Normal Forms (CNF), and then solving the CNF formulas with SAT solvers. Many CNF encodings have been proposed to handle different kinds of constraints, e.g. various CNF encodings for handling binary constraints [Kas90, IM94, Wal00, Gen02, VG08] and MDD constraints [AGMES16]. CNF encodings with SAT solvers have been shown to work very well on solving CSP instances. For example, the PicatSAT solver which uses CNF encodings and SAT solvers won the first place in the XCSP competition 2022[2] and the second place in the Minizinc challenge 2022.[3]

MDD, Regular and Table constraints are the most common ad-hoc constraints used in CSP solvers. In this thesis, we propose to improve the GAC propagators and CNF encodings of ad-hoc constraints from a binary encoding perspective. Figure 1.1 shows the framework of our methods. We use binary encodings to transform the MDD, Regular and Table constraints into binary constraints, and then handle the binary constraints with the AC propagators and CNF encodings of binary constraints. We highlight that it is not necessary to encode all constraints into binary constraints, i.e. the "other constraints" from Figure 1.1 can be non-binary (or global) with their own propagators and CNF encodings. A set of binary constraints can be regarded as a kind of global constraints, thus, the binary encodings of the ad-hoc constraints can work together with the GAC propagators and CNF encodings of the other constraints. Our results show that binary encodings can be more compact, and AC on the binary encodings can also be stronger than GAC on the original constraints, therefore, binary encoding methods can outperform the GAC propagators and CNF encodings of the original MDD, Regular and Table constraints.

[2] http://www.xcsp.org/competitions/

[3] https://www.minizinc.org/challenge.html

1.2 Research questions

The "folklore" on AC versus GAC has spurred major developments in algorithms for non-binary constraints. After 2010, most algorithms of solving CSPs directly deal with non-binary constraints. In this thesis, we will show that the "folklore" is misleading. It is not always the case that GAC propagators can outperform AC propagators with binary encodings. We will revisit the binary encodings of ad-hoc constraints and investigate why existing AC algorithms with binary encodings are slower than GAC algorithms on ad-hoc constraints. Moreover, we will try to answer the following 4 research questions.

(i) As shown in [SS05], the special structures of binary encodings can be used to design specialized AC propagators for the binary encodings. In addition, it is also possible to use new techniques from modern GAC algorithms to improve AC propagators. So the first research question we want to answer is that:

> *Is it possible to give a specialized AC algorithm, used to handle existing binary encodings such as the hidden variable encoding, which can be competitive with the state-of-the-art GAC algorithms of table constraints?*

(ii) By exploring the above research question in Chapter 5, we know that binary encoding methods have potential to outperform the state-of-the-art table GAC propagators. Then the next research question we try to answer is that:

> *Can we give better binary encodings to transform table constraints into binary constraints such that AC algorithms on the binary constraints can outperform the state-of-the-art GAC propagators of table constraints?*

(iii) To the best of our knowledge, all existing binary encodings of ad-hoc constraints are proposed to transform table constraints into binary constraints. As such the third research question we want to investigate is that:

> *Can we use binary encodings and AC propagators to improve the state-of-the-art GAC propagators of other ad-hoc constraints, such as the Ordered Multi-valued Decision Diagram (MDD) constraints?*

(iv) In Chapters 5-9, we show that binary encodings with specialized AC propagators can significantly outperform the GAC propagators of various ad-hoc constraints. Then the fourth research question we try to to explore is that:

> *In addition to AC propagators, can the other methods, such as the CNF encodings of binary constraints, work well on the binary encodings of various forms of ad-hoc constraints?*

1.3 Summary of contributions

We propose new binary encodings to transform various ad-hoc constraints into binary constraints, such as the table and MDD constraints. In addition, we give specialized AC propagators and CNF encodings to handle the binary encodings of ad-hoc constraints. We remark that although in principle all non-binary finite domain constraints can be solved with binary encodings, this may not be practical. Rather we use binary encodings to encode those ad-hoc constraints in a CSP model on which binary encoding methods can work very well, while the other non-binary constraints are directly handled with the appropriate non-binary methods, e.g., the best examples being global constraints which use specialized propagators.

1.3.1 An overview of the thesis

The thesis has 3 main parts. The first part gives basic notations of CSPs (Chapter 2) with background and surveys of ad-hoc constraints (Chapter 3) and binary encodings (Chapter 4). The second part including Chapters 5 and 6 focuses on the binary encodings of non-binary table constraints and specialized AC algorithms used to enforce AC on the binary encodings. The third part composed of Chapters 7-10 is about transforming decision diagram and automaton constraints into binary constraint trees. In addition, Chapter 11 summarizes the thesis. We now give outlines of the second and third parts.

Part II: Table constraints

- **Chapter 5: AC consistency revisited**

  1. We show some experimental results to analyze the weaknesses of well-known binary encodings and classic AC algorithms in solving non-binary CSPs.
  2. A new AC algorithm called HTAC is proposed to handle the HVE encoding.
  3. Experimental results show that HTAC with the HVE encoding can be competitive with the state-of-the-art GAC algorithms of table constraints.

- **Chapter 6: Bipartite encoding**

  1. A new binary encoding called Bipartite Encoding (BE) is proposed to encode non-binary table constraints into binary constraints.
  2. We give an algorithm to generate compact BE encodings.
  3. A specialized AC propagator called AC-BE, which exploits the structure of BE encodings, is proposed to enforce AC on the BE encodings.
  4. Experimental results show that the specialized AC propagator with BE encodings (AC-BE) can overall outperform the state-of-the-art GAC algorithms on the original non-binary table constraints.

Part III: Decision diagram and automaton constraints

- **Chapter 7: Binary constraint trees**

  1. We introduce a compact constraint representation called Binary Constraint Tree (BCT), which is a set of binary constraints with a tree structure.
  2. A binary encoding called direct tree binary encoding (DTBE) is proposed to encode non-binary decision diagram and automaton constraints into BCTs.
  3. We prove that the BCT constraints can be super-polynomially smaller than the MDD and non-deterministic finite state automaton (NFA) constraints.

- **Chapter 8: Reduction rules for BCT constraints**

  1. 4 reduction rules are proposed to simplify BCT constraints.
  2. We prove that it is NP-complete to construct the optimal BCT constraints by using the reduction rules to simplify the DTBE encoding.
  3. We propose an algorithm and a heuristic to simplify the DTBE encoding with the 4 reduction rules.
  4. Experimental results show that the reduction rules can significantly simplify the BCT constraints generated from the DTBE encoding.

- **Chapter 9: Enforcing AC on BCT Constraints**

  1. We propose a specialized AC propagator using various domain representations and revise functions to enforce AC on the BCT constraints.
  2. Experimental results on a large set of benchmarks show that our specialized AC propagator of BCT constraints can significantly outperform the state-of-the-art GAC algorithms of MDD constraints.

- **Chapter 10: CNF encodings of BCT Constraints**

  1. We introduce five CNF encodings to transform BCT constraints into CNF forms, making BCT constraints suitable for SAT solvers.
  2. We investigate and compare the strength of the unit propagation on the five CNF encodings of BCT constraints.
  3. Our experimental results show that the CNF encodings of BCT constraints can significantly outperform those of existing MDD constraints on various benchmarks.

1.3.2 Publications

Some results of this thesis have been published in the following papers:

- Ruiwei Wang and Roland H. C. Yap, Arc Consistency Revisited. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, 2019.
- Roland H. C. Yap, Wei Xia and Ruiwei Wang, Generalized Arc Consistency Algorithms for Table Constraints: A Summary of Algorithmic Ideas. In *AAAI National Conference on Artificial Intelligence (AAAI)*, 2020.
- Ruiwei Wang and Roland H. C. Yap, Bipartite Encoding: A New Binary Encoding for Solving Non-Binary CSPs. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.
- Ruiwei Wang and Roland H. C. Yap, Encoding Multi-valued Decision Diagram Constraints as Binary Constraint Trees. In *AAAI National Conference on Artificial Intelligence (AAAI)*, 2022.
- Ruiwei Wang and Roland H. C. Yap, CNF Encodings of Binary Constraint Trees. In *International Conference on Principles and Practice of Constraint Programming (CP)*, 2022.
- Ruiwei Wang and Roland H. C. Yap, The Expressive Power of Ad-hoc Constraints for Modelling CSPs. In *AAAI National Conference on Artificial Intelligence (AAAI)*, 2023.

Part A

Background and Literature Review

Chapter 2

Constraint Satisfaction Problems

2.1 Constraint Satisfaction Problems

It is well known that finite domain Constraint Satisfaction Problems (CSPs) are NP-complete. For example, the famous 3-SAT problem can be regarded as a special case of CSPs modelled with clause constraints over Boolean variables, and finite domain CSPs can be encoded as 3-SAT instances with CNF encodings [Pre09]. In addition, a wide range of combinatorial problems from Artificial Intelligence and Operation Research can be modelled as CSPs, e.g., many scheduling and planning problems [FS90, DK01] can be modelled as CSPs. In this section, we describe some basic notations and definitions of finite domain CSPs that will be used in the rest of this thesis.

Definition 2.1. *A* CSP *instance P is a pair (X, C) such that*

- *X is a set of variables and C is a set of constraints and $\mathcal{D}(x)$ denotes the* domain *of a variable $x \in X$, which is a set of values, and*
- *a* literal *of a variable $x \in X$ is a variable value pair (x, a) and a* tuple *over r variables $\{x_{i_1}, \cdots, x_{i_r}\}$ is a set of literals $\{(x_{i_1}, a_1), \cdots, (x_{i_r}, a_r)\}$, and*
- *each constraint $c \in C$ consists of a* constraint scope *$scp(c)$ and a* constraint relation *$rel(c)$ where $scp(c) \subseteq X$ and $rel(c)$ is a set of tuples over $scp(c)$.*

We say a variable x is included in a constraint c if $x \in scp(c)$, and a variable x (or a constraint c) is included in a CSP (X, C) if $x \in X$ (or $c \in C$). Then a variable is a *finite domain variable* if its domain is finite. A constraint c is a *finite domain constraint* if all variables included in c are finite domain variables. A CSP (X, C) is a *finite domain CSP* if all variables in X are finite domain variables. A CSP can be modelled with various constraints such as ad-hoc constraints (also called generic constraints) and special purpose global constraints, where *ad-hoc constraints* denote the constraints which can have arbitrary constraint relations. A large body of constraint representations have been used to define ad-hoc constraints, e.g., the table [BR97], automaton [Pes04] and decision diagram [CY10] constraints. In this thesis, we will focus on finite domain CSPs and ad-hoc constraints.

For each constraint c, the number of variables in the constraint scope $scp(c_i)$ is called the *(constraint) arity* of c. Then an *unary constraint* is a 1-arity constraint. A *binary constraint* is a 2-arity constraint. A *ternary constraint* is a 3-arity constraint. A *non-binary constraint* is a k-arity constraint such that $k > 2$. A *binary CSP* is a CSP such that all constraints included in the CSP are binary constraints. A *non-binary CSP* is a CSP including at least one non-binary constraint. A CSP (X, C) is a *normalized CSP* if $scp(c_i) \nsubseteq scp(c_j)$ and $scp(c_j) \nsubseteq scp(c_i)$ for any two constraints $c_i, c_j \in C$. Without loss of generality, we assume that all unary constraints c included in a CSP are removed by updating the domain of the variable $x \in scp(c)$ as the set of values $\{a \in \mathcal{D}(x) | \{(x, a)\} \in rel(c)\}$.

A *c-tuple* τ over a set of variables V is a set of literals such that for each $(x, a) \in \tau$, the variable x is in V. Given a c-tuple τ over variables V and a set T of c-tuples over V, we use $\tau[V] = \{(x, a) \in \tau | x \in V\}$ to denote a subset of τ, while $T[V] = \{\tau[V] | \tau \in T\}$ is the *projection* of T on V. We highlight that any tuple is also a c-tuple and a c-tuple can be used to denote current variable domains. Then $P_{|\tau}$ denotes a sub-problem of a CSP $P = (X, C)$, where τ is a c-tuple, which is generated by removing the values a from the variable domains $\mathcal{D}(x)$ such that $x \in X$ and $\tau[\{x\}] \neq \emptyset$ and $(x, a) \notin \tau[\{x\}]$. $P_{|x=a}$ denotes a sub-problem generated by assigning a value a to a variable x in a CSP P, i.e., $P_{|x=a}$ is equal to $P_{|\{(x,a)\}}$. In addition, $P_{|x\neq a}$ is used to denote the sub-problem $P_{|\{(x,b)|b\in\mathcal{D}(x),a\neq b\}}$.

A c-tuple τ is *valid* if $a \in \mathcal{D}(x)$ for all literals $(x, a) \in \tau$. A tuple τ is an *assignment* if τ is valid. A tuple τ over a set of variables S is *consistent* on a CSP (X, C) if $\tau[scp(c)] \in rel(c)$ for all constraints $c \in C$ such that $scp(c) \subseteq S$. A tuple τ over a set of variables X is a *solution* of a CSP $P = (X, C)$ if τ is an assignment and τ is consistent on P. $sol(X, C)$ (and $sol(P)$) is used to denote the solution set consisting of all solutions of a CSP $P = (X, C)$. Then the CSP P is *satisfiable* if $Sol(X, C) \neq \emptyset$, otherwise it is *unsatisfiable*. Solving a CSP instance P is to check whether P is satisfiable.

A variable x is called a *Boolean variable* if its variable domain $\mathcal{D}(x)$ is a subset of $\{true, false\}$. A constraint c over a set of r Boolean variables $\{x_{i_1}, \cdots, x_{i_r}\}$ is called a *clause* if the constraint is equal to the disjunction of a set of literals $cl = \{(x_{i_1}, a_1), \cdots, (x_{i_r}, a_r)\}$ such that $a_j \in \{True, False\}$ for all $1 \leq j \leq r$, where the constraint relation $rel(c)$ consists of the tuples τ over $scp(c)$ such that τ includes at least a literal in cl, i.e., $\tau \cap cl \neq \emptyset$. A CSP (X, C) is called a *Conjunctive Normal Form (CNF)* if all variables in X are Boolean variables and all constraints in C are clauses.

Assume (X, C) is a CSP and x is a variable in X. Then we use $NC(x) = \{c \in C | x \in scp(c)\}$ to denote the set of constraints in C whose constraint scopes include the variable x. Additionally, $N^-(x) = N(x) \setminus \{x\}$ is used to denote the variable set which includes all neighbors of x, and $N(x) = \cup_{c\in NC(x)} scp(c)$ is used to denote the variable set which includes the variable x and all neighbors of x.

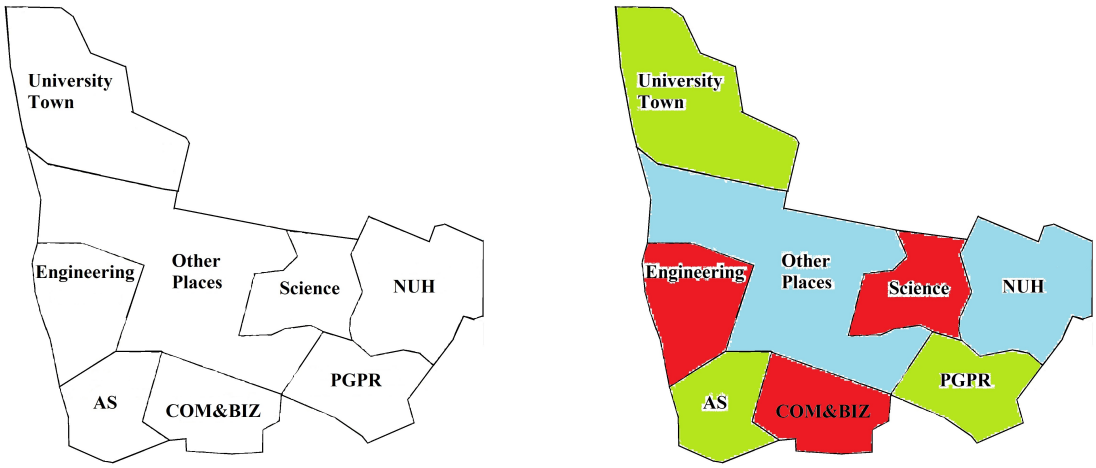


Figure 2.1: Coloring the map of NUS Kent Ridge Campus.

Example 2.1. *Graph coloring problems can be modelled as CSPs. Figure 2.1a shows a map of the NUS Kent Ridge Campus. Coloring all 8 places in the map with 3 colors* $\{Red, Green, Blue\}$ *is modelled as a CSP* (X, C) *such that*

- X *is equal to* $\{x_1, \cdots, x_8\}$ *where every variable denotes a place and the variable domain is the set of colors* $\{Red, Green, Blue\}$*, and*
- *the variables* $x_1, \cdots, x_8$ *respectively denote the places University Town, Other Places, Science, Engineering, AS, COM&BIZ, PGPR and NUH in the map, and*
- C *is equal to the set* $\{x_1 \neq x_2, x_2 \neq x_3, x_2 \neq x_4, x_2 \neq x_5, x_2 \neq x_6, x_2 \neq x_7, x_3 \neq x_7, x_3 \neq x_8, x_4 \neq x_5, x_5 \neq x_6, x_7 \neq x_8\}$ *of binary inequalities, where for any two variables* $x, y \in X$ *and an inequality constraint* $x \neq y$ *between the two variables* x *and* y*, the constraint relation of* $x \neq y$ *is equal to* $\{\{(x, a), (y, b)\} | a \in \mathcal{D}(x), b \in \mathcal{D}(y), a \neq b\}$ *and the constraint scope of* $x \neq y$ *is equal to* $\{x, y\}$*.*

Every solution of the CSP corresponds to a feasible coloring of the map. For each variable $x_i \in X$ *and a neighbor* $x_j \in N(x_i)$*, the constraint* $x_i \neq x_j$ *is included in* C*, thus,* x_i *and* x_j *are assigned different colors in the solution. Figure 2.1b is a coloring corresponding to the solution* $\{(x_1, Green), (x_2, Blue), (x_3, Red), (x_4, Red), (x_5, Green), (x_6, Red), (x_7, Green), (x_8, Blue)\}$*. This CSP is a finite domain binary CSP, since all constraints in the CSP are finite domain binary constraints and all variables in the CSP are finite domain variables.*

2.2 Constraint graphs

The structure of CSPs can be described by constraint graphs. Generally, the complexity of a CSP is related to its structures. For example, a CSP whose constraint graph is a tree or has bounded treewidth can be solved in polynomial time [Fre82, DP87]. Many kinds of constraint graphs have been proposed to describe the structure of CSPs [Dec92], such as the primal (constraint) graph and the dual (constraint) graph. In this thesis, we will use the primal graph as the default constraint graph of a CSP unless we emphasize that the constraint graph used is other graphs.

2.2.1 Primal graph

Initially, binary constraint graphs are used to describe the structure of binary CSPs [Fre82], where the *binary constraint graphs* regard variables and binary constraints as nodes and edges, respectively. Then primal graphs are proposed to generalize the binary constraint graphs from binary CSPs to non-binary CSPs [DP88, DP89]. The *primal graph* of a CSP is an undirected graph such that the variables in the CSP are nodes and there is an edge between two nodes (variables) x and y in the graph if the CSP has a constraint c including the two variables x and y, i.e., $\{x, y\} \subseteq scp(c)$. A primal graph is an *acyclic graph* if there is not any cycle in the primal graph, and an acyclic graph is a *tree* if the number of edges in the graph is equal to the number of nodes minus 1.

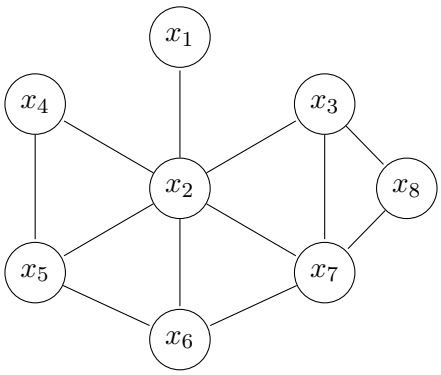


Figure 2.2: A primal graph.

Example 2.2. *Figure 2.2 shows the primal graph of the CSP given in Example 2.1. We can see that the nodes in the primal graph are the variables in the CSP. In addition, any two variables are connected in the primal graph if there is a constraint including the two variables in the CSP, e.g., the two variables x_3 and x_7 are connected in the primal graph due to the constraint $x_3 \neq x_7$ includes both the variables x_3 and x_7. Obviously, the primal graph given in Figure 2.2 is not an acyclic graph, as it includes cycles.*

2.2.2 Dual graph

In addition to the primal graph, the dual graph is another constraint graph which generalizes the binary constraint graph from binary CSPs to non-binary CSPs. A motivation behind the dual graph is to transform non-binary CSPs into binary CSPs by regarding constraints as *c-variables* [DP89] (also called dual variables in [SW99b]), and then the binary constraint graph over the c-variables is used to describe the structure of non-binary CSPs.

Formally, the *dual graph* [DP88, DP89] of a CSP (also called intersection graph in database theory [Mai83]) is an undirected graph such that the nodes in the dual graph are the constraints in the CSP and there is an edge in the graph between 2 nodes (constraints) c_i and c_j if the constraint scopes of c_i and c_j share at least one variable, i.e., $scp(c_i) \cap scp(c_j) \neq \emptyset$.

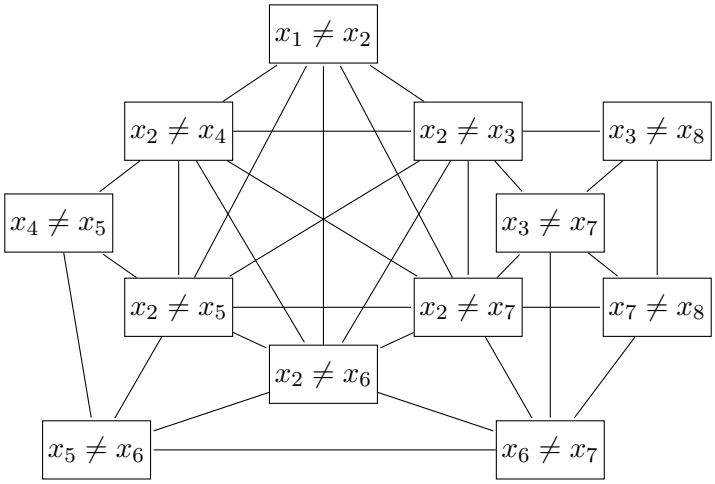


Figure 2.3: A dual graph.

Example 2.3. *Figure 2.3 is the dual graph of the CSP given in Example 2.1, where the constraints in the CSP are regarded as the nodes in the dual graph. For each two connected nodes (constraints) in the graph, the constraint scopes of the two nodes share a variable. For example, the constraints* $x_1 \neq x_2$, $x_2 \neq x_3$, $x_2 \neq x_4$, $x_2 \neq x_5$, $x_2 \neq x_6$, $x_2 \neq x_7$ *are connected to each other, since their constraint scopes share the variable* x_2.

The dual graph of a CSP corresponds to a binary CSP, i.e., the dual encoding (see Chapter 4), where each edge in the graph denotes a binary constraint in the binary CSP, and the edge is *redundant* if eliminating the binary constraint does not change the solutions of the binary CSP [DD87]. Then by removing redundant edges from the dual graph, we can get a new constraint graph called *the join graph*. Similar to the primal graph, if the join graph of a CSP is a tree, then the CSP can be solved in polynomial time, where the tree structural join graph is called *a join tree* [DP89].

2.3 Constraint propagation

Constraint propagators (also called propagation algorithms) are key components used in Constraint Programming (CP)/CSP solvers. Usually, the constraint propagators are used to enforce local consistencies on CSPs to remove values from variable domains or tuples from constraint relations. The local consistencies can be tracked back to the seminal work of Mackworth in 1977 [Mac77a] about arc consistency (AC) and path consistency. Since then there has been a large body of research on consistencies, such as the generalized arc consistency (GAC) [Mac77b, MM88] and pairwise consistency (PWC) [JJNV89]. A motivation behind the local consistencies (in particular GAC) is that an assignment can be removed from a CSP if it is not included by any solution of the CSP, in other words, eliminating the assignments which are not globally consistent from the CSP.

Definition 2.2 (Global consistency)**.** *A tuple is* globally consistent *on a CSP $P = (X, C)$ if it is included by a solution of P. The CSP P is* globally consistent *if the domains of all variables in X are not empty and every tuple τ over a subset of X which is valid and consistent on P is globally consistent on P.*

It is NP-hard to check whether a tuple is globally consistent on CSPs, since CSPs are NP-complete. To address this issue, many local consistencies, i.e., sufficient conditions of global consistency, are proposed [Bes06]. The local consistencies can be categorized as domain consistency which eliminates values from variable domains and relation consistency which eliminates tuples from constraint relations. In this section, we introduce the local consistencies used in the thesis.

2.3.1 Generalized Arc Consistency

Generalized Arc Consistency (GAC) is a local domain consistency which has been applied in most CP/CSP solvers, such as Choco [JRL08], Gecode [STL13] and OR-Tools[1].

Definition 2.3 (GAC)**.** *A tuple $\tau \in rel(c)$ is a* support *of a variable value $a \in \mathcal{D}(x)$ and a literal (x, a) if $(x, a) \in \tau$. A variable value $a \in \mathcal{D}(x)$ is* generalized arc consistent *(GAC) on c if a has a valid support in $rel(c)$. A variable x is GAC on c if $\mathcal{D}(x) \neq \emptyset$ and all values $a \in \mathcal{D}(x)$ are GAC on c. A constraint c is GAC if all variables in $scp(c)$ are GAC on c. A CSP (X, C) is GAC if all constraints in C are GAC.*

Enforcing GAC on a CSP P is to find a maximum valid c-tuple τ such that the sub-problem $P_{|\tau}$ is GAC. We can enforce GAC on a constraint c by removing the inconsistent variable values which are not GAC on c from variable domains. Especially, we can use different GAC algorithms to remove inconsistent variable values for different constraint types.

[1] https://developers.google.com/optimization/cp/

The constraint relations of special purpose global constraints are restricted to specific sets of tuples over constraint scopes. Therefore, it often has efficient specialized algorithms to enforce GAC on special purpose global constraints by exploiting their semantics. For example, maximum bipartite matching algorithms can be used to enforce GAC on the AllDifferent constraint [Rég94]. However, this is also a weakness of special purpose global constraints, i.e., there are constraint relations which cannot be easily modelled with well studied special purpose global constraints.

Unlike special purpose global constraints, ad-hoc constraints (also known as generic constraints) can be used to model arbitrary constraint relations as various representations. Many efficient GAC algorithms using different constraint representations have been proposed to enforce GAC on ad-hoc constraints. For example, the table GAC algorithms, such as the simple tabular reduction (STR) [Ull07, Lec11], STRbit [WXYL16] and compact-table (CT) algorithms [DHL+16], can be used to enforce GAC on table constraints.

Example 2.4. *All constraints of the CSP given in Example 2.1 are GAC. For example, the value Red in the domain of the variable U has a valid support,* $\{(x_1, Red), (x_2, Blue)\}$*, on the constraint* $x_1 \neq x_2$*, thus, Red is GAC on the constraint* $x_1 \neq x_2$*.*

2.3.2 Arc Consistency

Historically, many new techniques were firstly studied on binary constraints and then extended to non-binary constraints. For example, GAC is generalized from the original Arc Consistency (AC) [Mac77a] on binary CSPs.

Definition 2.4 (AC)**.** *A value* $a \in \mathcal{D}(x)$ *is AC on a binary constraint* c *where* $scp(c) = \{x, y\}$ *if there is a value* $b \in \mathcal{D}(y)$ *such that* $\{(x, a), (y, b)\} \in rel(c)$*. A variable* x *is AC on* c *if* $\mathcal{D}(x) \neq \emptyset$ *and all values in* $\mathcal{D}(x)$ *are AC on* c*. A binary constraint* c *is AC if all variables in* $scp(c)$ *are AC on* c*. A binary CSP is AC if all binary constraints in the CSP are AC.*

AC is a special case of GAC on binary CSPs. Many AC algorithms have been extended to enforce GAC on non-binary constraints. For example, the AC3 [Mac77a] and AC4 [MH86] algorithms are extended to the GAC3 [Mac77b] and GAC4 [MM88] algorithms.

2.3.3 Full Pairwise Consistency

Moreover, in order to improve the strength of GAC, some high-order consistencies are proposed to combine GAC with local relation consistencies. For example, the GAC and Pairwise Consistency (PWC) [JJNV89] are combined with the Full Pairwise Consistency (FPWC) [LPS13, LXY14, LXY15].

Definition 2.5 (FPWC)**.** *A tuple* $\tau' \in rel(c_j)$ *is a* support *of a tuple* $\tau \in rel(c_j)$ *iff* $\tau[S] = \tau'[S]$ *where* $S = scp(c_i) \cap scp(c_j)$*. A tuple* $\tau \in rel(c_i)$ *is PWC on* c_j *if* τ *has a valid*

support in $rel(c_j)$. *A constraint* c_i *is PWC on* c_j *if all tuples* $\tau \in rel(c_i)$ *are PWC on* c_j. *A pair of constraints* c_i, c_j *is PWC if* c_i *is PWC on* c_j *and* c_j *is PWC on* c_i. *A CSP* (X, C) *is PWC if all pairs of constraints in* C *are PWC. A CSP* P *is FPWC if* P *is PWC and GAC.*

The consistency level of FPWC can be stronger than GAC, since FPWC maintains both PWC and GAC at the same time, thus, all FPWC CSP instances are also GAC.

Example 2.5. *All pairs of constraints are PWC for the CSP* P *given in Example 2.1. For example, the tuple* $\{(x_1, Red), (x_2, Blue)\}$ *in the relation of* $x_1 \neq x_2$ *has a valid support* $\{(x_2, Red), (x_3, Blue)\}$ *in the relation of* $x_2 \neq x_3$, *thus, the tuple* $\{(x_1, Red), (x_2, Blue)\}$ *is PWC on the constraint* $x_2 \neq x_3$. *In addition,* P *is GAC, therefore,* P *is FPWC.*

2.3.4 GAC propagation algorithms

Usually, the constraints in a CSP (X, C) may affect each other. A constraint in C which is already GAC may become not GAC after enforcing GAC on another constraint in C. GAC propagation algorithms are used to iteratively enforce GAC on the constraints in C until all constraints in C are GAC, i.e., achieving a fixed point [Bes06].

Algorithm 2.1 is a propagation algorithm that enforces GAC on a CSP (X, C). The algorithm uses a propagation queue $\mathcal{Q}$ to record all variables $x \in X$ such that x may not be GAC on a constraint which includes x. $\mathcal{Q}$ is initialized as X, and between Lines 3-9, the algorithm selects a variable x from $\mathcal{Q}$, and then enforces GAC on all constraints including x. We can use different propagation heuristics [BHL04] to select a variable x from $\mathcal{Q}$, e.g., selecting the variable with minimum domain size. The algorithm returns false if there exists

Algorithm 2.1: GAC(X, C)

```
1  Q ← X
2  while Q ≠ ∅ do
3      pick and delete variable x from Q
4      for c ∈ C s.t. x ∈ scp(c) do
5          if ¬enforceGAC(c) then return false
6          for y ∈ scp(c) do
7              if D(y) is changed then Q ← Q ∪ {y}
8          end
9      end
10 end
11 return true
```

a constraint which is not GAC. The algorithm *enforceGAC(c)* is called to enforce GAC on a constraint c by removing all inconsistent values from variable domains. Note that for different constraint types, we can implement different *enforceGAC(c)* algorithms. For example, we can use flow-based algorithms to handle global cardinality constraints [Rég96]. For each variable $y \in scp(c)$, all values which are not GAC on the constraint c are removed from $\mathcal{D}(y)$ (at Line 5). If the domain $\mathcal{D}(y)$ is updated, then y is added into $\mathcal{Q}$. If $\mathcal{D}(y) = \emptyset$, then the constraint is not GAC and the function returns false.

2.3.5 Unit propagation

Unit propagation [DP60] is the GAC propagator used to enforce GAC on the clause constraints in a CNF F. We use $UP(F)$ to denote the CNF generated from F by removing all variable values which are not GAC on a clause in F. If $UP(F)$ includes any variable with empty domain, then F is unsatisfiable. Every clause c in F is a disjunction of the literals in a tuple cl, and we can enforce GAC on c by removing the variable values $a \in \mathcal{D}(y)$ such that $y \in scp(c)$ and $(y, a) \notin cl$ and for all $(x, b) \in cl$, $b \notin \mathcal{D}(x)$ or $x = y$.

2.4 MGAC Search algorithms

GAC propagator and backtracking search are two important techniques used for solving CSPs, where backtracking search is used to try all assignments over X, and GAC propagators are used to reduce search space and check whether an assignment is a solution. Maintaining GAC during backtracking search (MGAC) is a good strategy for solving CSPs [SF94].

Algorithm 2.2 gives the details of a MGAC algorithm. At Line 2, the algorithm calls the GAC propagation algorithm to maintain GAC on a CSP instance P=(X,C). If P is not GAC, the algorithm uses backtracking search (between Lines 4-7) to check whether P is

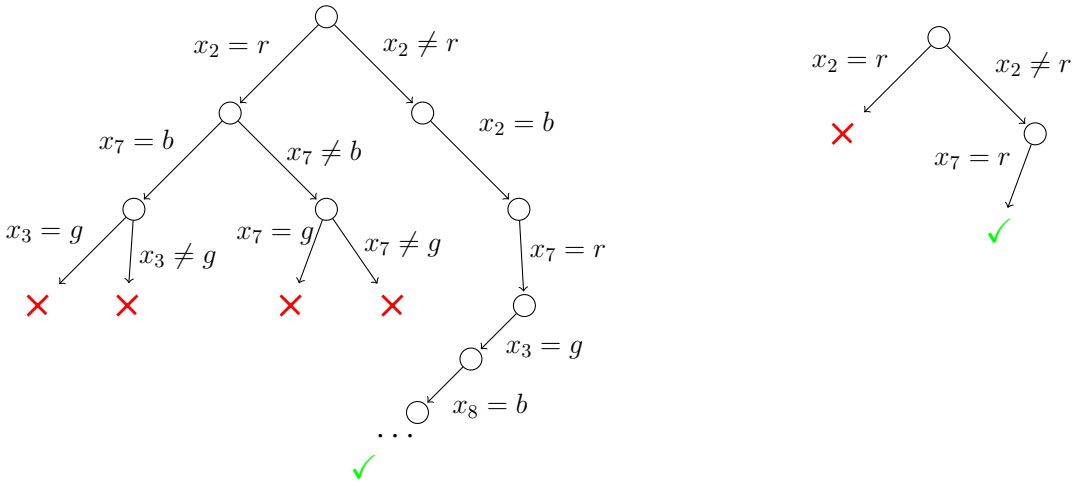
Algorithm 2.2: MGAC (P)

1 $(X, C) \leftarrow P$
2 **if** $\neg Propagation(X, C)$ **then** **return** false
3 **if** $\forall x \in X$ *s.t.* $|\mathcal{D}(x)| = 1$ **then** **return** true
4 select a variable x from X such that $|\mathcal{D}(x)| > 1$
5 select a value a from $\mathcal{D}(x)$
6 **if** $MGAC(P_{|x=a})$ **then** **return** true
7 **return** $MGAC(P_{|x\neq a})$

satisfiable. At Lines 4 and 5, the algorithm uses heuristics, such as the wdeg/dom[BHLS04], impact[Ref04], activity [MVH12] and correlation [WXY17] heuristics, to select a variable x and a value $a \in \mathcal{D}(x)$ such that the domain $\mathcal{D}(x)$ has at least 2 values. Then the value a is assigned to x, and recursively using Algorithm 2.2 to solve the sub-problem $P_{|x=a}$. If $P_{|x=a}$ is satisfiable, then P is satisfiable. Otherwise Algorithm 2.2 is used to solve another sub-problem $P_{|x\neq a}$. Then P is satisfiable iff $P_{|x\neq a}$ is satisfiable, since $P_{|x=a}$ is not satisfiable.

In addition, there are many techniques which can be used to improve the MGAC algorithm, such as the search restart technique [LSTV07]. If the MGAC algorithm selects a branch, i.e., selecting a variable and a value at Lines 4 and 5, which leads to a very large search tree, then the algorithm may be stuck in the large search tree. In order to resolve this issue, we can set a cutoff for the number of allowed failed assignments and restart the search algorithm when it exceeds the cutoff. The information used by various search heuristics are dynamically updated during the search, thus, the search algorithm may have a better search tree after restarting. Many restart policies can be used to set the cutoff number, such as the Luby [LSZ93], geometrical [Wal99] and dynamic [KHR+02] policies.

Example 2.6. *For the CSP given in Example 2.1, we construct a new instance by setting the variable domains as:* $\mathcal{D}(x_2) = \{r, b\}$, $\mathcal{D}(x_3) = \mathcal{D}(x_6) = \{r, g\}$, $\mathcal{D}(x_4) = \mathcal{D}(x_5) = \mathcal{D}(x_7) = \{r, b, g\}$, $\mathcal{D}(x_1) = \mathcal{D}(x_8) = \{b, g\}$. *Figure 2.4 shows the search trees of solving the instance by the MGAC algorithm and another algorithm using a simple technique without GAC which removes the inconsistent values in the neighbors of a variable x after assigning a value to x during backtracking search. Figure 2.4a is a search tree of the method without GAC, and Figure 2.4b is the search tree of MGAC. We can see that MGAC has much less search nodes than the one without GAC. MGAC skips those variables whose domain sizes are 1, thus, it can solve the problem after assigning 2 values to the variables x_2 and x_7.*



(a) No GAC **(b)** MGAC

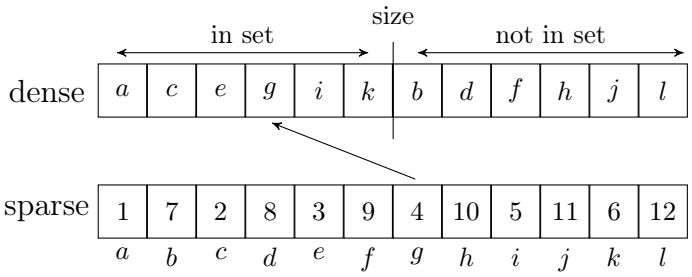
Figure 2.4: Search trees comparison.

2.5 Variable domain representations

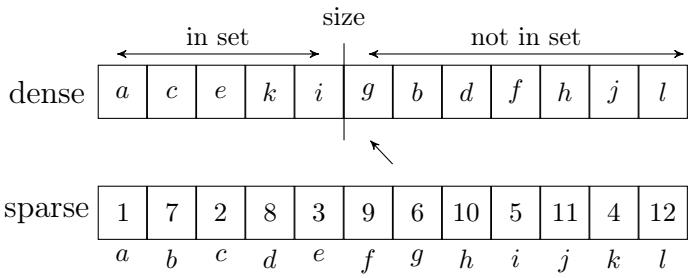
Many set representations can be used in CSP solvers to represent variable domains. By using different variable domain representations, we can have different AC algorithms. For example, the AC5 algorithm uses the ordered linked set [VHDT92, LS06] variable domains, and the AC3^{bit} algorithm represents variable domains as both ordered linked sets and bit sets [SC06, LV08]. In this section, we introduce four domain representations: sparse set [BT93, dSMSSL13], ordered linked set, bit set and sparse bit set [DHL+16]. These representations are used in the AC algorithms proposed in this thesis.

2.5.1 Sparse set

A sparse set has three components: two arrays and a number denoting the size of the set. Figure 2.5a shows a sparse set which represents a subset S of $\{a, b, \cdots, l\}$. The size of S is 6. The "dense" array records all elements in $\{a, b, \cdots, l\}$ and the subset S consists of the first 6 elements in dense. The "sparse" array records the position of each element in dense, e.g., the position of the element g is 6. It is easy to delete an element from a sparse set, and restore the deleted elements. Figure 2.5b shows the result of removing g from S. It exchanges g with k which is the 6^{th} element in dense, and then decreases the size to 5. The sparse array is also changed after updating the dense array. In addition, the old size of S is recorded in a stack, and S can be directly restored by increasing its size back to the old size 6.



(a) Sparse set



(b) Removing element g from the sparse set

Figure 2.5: Sparse sets.

2.5.2 Ordered linked set

An ordered linked set consists of two arrays and a doubly linked list and a singly linked list. Figure 2.6 shows an ordered linked set for a subset S of $\{a, \cdots, l\}$. The "value" array records all elements in $\{a, \cdots, l\}$, and a doubly linked list sorted in increasing order is used to link the elements in S. The "absent" array indicates which elements are included in S. $absent[m]$ is equal to "No" if the m^{th} element is included in S, otherwise $absent[m]$ records when the m^{th} element is removed from S. A singly linked list links the elements which are not in S with the reverse order of deleting elements, e.g., f, h are the last 2 elements deleted from S, so f, h are the first 2 elements in the singly linked list. Then we can restore S by adding the elements in the singly linked list back to S in the reverse order of deleting elements.

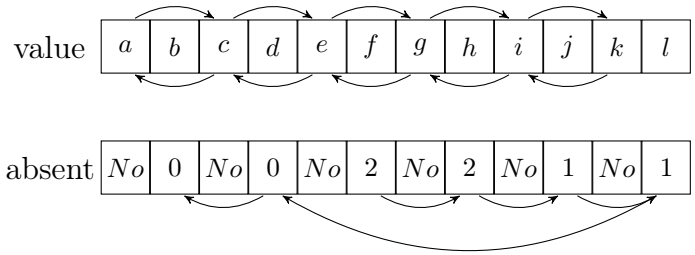


Figure 2.6: An ordered linked set.

2.5.3 Bit set

For a subset of a set, we can use a set of bits to denote whether an element is in the subset. Every bit corresponds to an element, where 1 means the element is in the subset and 0 is not. A bit set consists of a set of words, where each word includes w bits. Figure 2.7 shows a bit set which represents a subset S of $\{a, b, ...l\}$. 4 bits in the w_3 word are 1, and the bits in other words are 0, which means that S is the last 4 elements $\{i, j, k, l\}$. In order to restore subsets, we need to record the old status of a word if the word is updated. For example, w_3 becomes 0011 after removing i and j from S, and the old status of w_3 is 1111. Therefore, $w_3 = 1111$ is recorded in a stack before updating w_3.

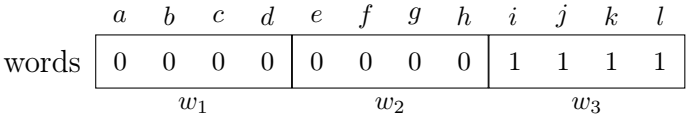


Figure 2.7: A bit set.

2.5.4 Sparse bit set

A sparse bit set has three components: a bit set, a dense array and a size which means the number of non-ZERO words in the bit set, where a word is ZERO if it only includes 0 bits, otherwise it is a non-ZERO word. Then the dense array is used to record the non-ZERO words. In Figure 2.8, the words w_1 and w_2 only include 0 bits. Therefore, the size of the dense array is 1 and the word w_3 is the first element in the dense array. If a word (or the size of non-ZERO words) is changed, then the old status is recorded in a stack which can be used to restore the sparse bit set.

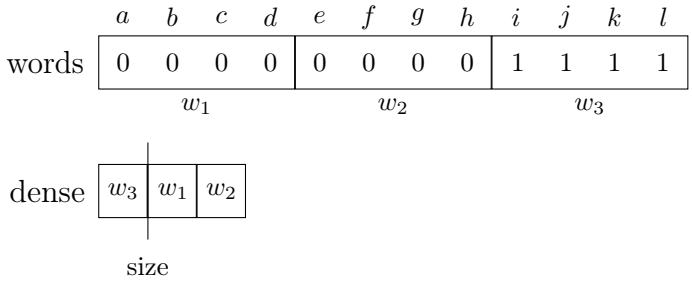


Figure 2.8: A sparse bit set.

Chapter 3

Ad-hoc constraints

3.1 Introduction

Usually, CSPs can be modelled with different constraints (depending on the problem modelled), e.g., the pigeonhole problem can be modelled as an AllDifferent constraint or a set of binary inequality constraints or a set of binary table constraints. We will focus on the ad-hoc constraints (also called generic constraints) which have efficient GAC algorithms,[1] where ad-hoc constraints can be used to model arbitrary constraint relations and provide the most modelling flexibility. Many representations have been proposed to define ad-hoc constraints, such as table [BR97], automaton [Pes04, QW06], context-free grammar [Sel06, QW06], non-ordinary table [KW07, JN13, MDL15] and decision diagram [CY10, VLS18] constraints. Ad-hoc constraints are very useful for modelling the specific constraints which do not fit well with special purpose global constraints. For example, the patterned stretch constraints used in the rostering problem can be modelled as regular constraints [Pes04]; the transition constraints used in planning problems can be encoded as short table constraints [GNNS15]; and the decision trees and random forest embedded in a CSP model can be encoded as table and c-table constraints [BLM15].

The table constraint, regular constraint [Pes04] and Ordered Multi-valued Decision Diagram (MDD) constraint [CY08, CY10] are three commonly used ad-hoc constraints. These ad-hoc constraints are supported by various Constraint Programming (CP) modelling tools and languages, such as Minizinc [NSB+07], Essence [FHJ+08] and XCSP [BLAP16]. Furthermore, many GAC algorithms of the table constraint, MDD constraint and regular constraint have been implemented in different CSP solvers, such as the SICStus Prolog [CWA+88], Abscon [MLB01] and Gecode [STL13] solvers, where the regular constraint is encoded as MDD constraints in the Abscon solver.[2] In this thesis, we will try to improve the GAC algorithms of ad-hoc constraints by transforming them into binary constraints and using AC algorithms to enforce AC on the binary constraints.

[1] Intensional constraints are also generic constraints but enforcing GAC on them is NP-hard.

[2] The experimental results given in [CY10, PR15] show that the GAC algorithms of MDD constraints can outperform those of regular constraints.

3.2 Definitions and notations

In this section, we introduce basic definitions and notations used for the table, c-table [KW07], Ordered Multi-valued Variable Diagram (MVD) [AFNP14], MDD, regular, and NFA (non-deterministic finite state automaton) constraints, where MVD is also called non-deterministic MDD. The basic definitions and notations will also be used in the other chapters of this thesis. These are ad-hoc constraints, since they can have arbitrary constraint relations over finite domain variables.

3.2.1 The table constraint

A canonical way of modelling a finite domain constraint is simply to represent the constraint relations as tables, where a *table* is a set of tuples. The term table constraint is used to denote the constraints represented as tables. Table constraints can be viewed as a general way of defining constraints, since the definition of constraint relation is also a set of tuples (see Chapter 2). Historically, many techniques of solving CSPs are designed with table constraints, such as the constraint relation synthesis algorithm [Fre78], the GAC4 algorithms [MM88], the adaptive consistency (or bucket elimination) algorithm [DP87, Dec99], the optimal k consistency algorithm [Coo89] and the PWC algorithm [JJNV89].

Example 3.1. *Figure 3.1 gives an example of table constraints. The table consists of 7 tuples* $\tau_0, \cdots, \tau_6$. *Each tuple is a set of literals, e.g.,* τ_1 *is the literal set* $\{(x_0, 0), (x_1, 0), (x_2, 1)\}$ *and* τ_6 *is the literal set* $\{(x_0, 1), (x_1, 2), (x_2, 2)\}$. *We remark that for each variable in the constraint scope, a tuple has exactly one literal of the variable.*

Assume the current variable domains are $\{0, 1, 2\}$. *Then the value* 2 *in the current domain of the variable* x_0 *is not GAC on the table constraint, since the literal* $(x_0, 2)$ *is not included by any valid tuple in the table.*

τ_0	$(x_0, 0)$	$(x_1, 0)$	$(x_2, 0)$
τ_1	$(x_0, 0)$	$(x_1, 0)$	$(x_2, 1)$
τ_2	$(x_0, 0)$	$(x_1, 1)$	$(x_2, 0)$
τ_3	$(x_0, 0)$	$(x_1, 1)$	$(x_2, 1)$
τ_4	$(x_0, 1)$	$(x_1, 2)$	$(x_2, 0)$
τ_5	$(x_0, 1)$	$(x_1, 2)$	$(x_2, 1)$
τ_6	$(x_0, 1)$	$(x_1, 2)$	$(x_2, 2)$

Figure 3.1: A table constraint.

3.2.2 The c-table constraint

All constraint relations over finite domain variables can be regarded as tables but the number of tuples in the tables may be exponential in constraint arity. Therefore, a number of new compact table representations have been proposed. We call such tables: non-ordinary tables. Katsirelos and Walsh proposed the first non-ordinary table called compressed table (c-table) [KW07] which can be exponentially smaller than table for modelling some constraints. For example, the negation of AllDifferent constraints and the Disjunctive Normal Form (DNF) can be represented as polynomial size c-tables but not tables.

A *compressed tuple (c-tuple)* τ over a set of variables V is a set of literals such that for each literal $(x, a) \in \tau$, the variable x is included in V (a definition from [JN13]). More details about c-tuple can also be found in Chapter 2. Every c-tuple τ over variables V represents a set of tuples denoted as $\mathcal{T}(\tau) = \{\tau' \subseteq \tau | \tau' \text{ is a tuple over } V\}$. In addition, a c-tuple can also be regarded as a Cartesian product of sets of variable values (a definition from [KW07]), i.e., $\mathcal{T}(\tau)$ can be encoded as $\prod_{x \in V}\{(x, a)|(x, a) \in \tau\}$. A *compressed table (c-table)* is a set of c-tuples. A c-table rel_1 represents a set of tuples rel_2 if the c-tuples in rel_1 represent the tuples in rel_2, i.e., $\cup_{\tau \in rel_1}\mathcal{T}(\tau) = rel_2$

Example 3.2. *Figure 3.2 shows a c-table encoding the tuples in the table given in Figure 3.1. The c-table consists of 2 c-tuples c-τ_0 and c-τ_1. Each c-tuple can be represented as both a set of literals (see Figure 3.2b) and a Cartesian product of sets of literals (see Figure 3.2a), e.g., the c-tuple c-τ_0 can be defined as the set* $\{(x_0, 0), (x_1, 0), (x_1, 1), (x_2, 0), (x_2, 1)\}$ *and also the Cartesian product* $\{(x_0, 0)\} \times \{(x_1, 0), (x_1, 1)\} \times \{(x_2, 0), (x_1, 2)\}$*, which can represent 4 tuples* $\{(x_0, 0), (x_1, 0), (x_2, 0)\}$, $\{(x_0, 0), (x_1, 0), (x_2, 1)\}$, $\{(x_0, 0), (x_1, 1), (x_2, 0)\}$ *and* $\{(x_0, 0), (x_1, 1), (x_2, 1)\}$.

c-τ_0	$\{(x_0, 0)\} \times \{(x_1, 0), (x_1, 1)\} \times \{(x_2, 0), (x_1, 2)\}$
c-τ_1	$\{(x_0, 1)\} \times \{(x_1, 2)\} \times \{(x_2, 0), (x_2, 1), (x_2, 2)\}$

(a) Cartesian Product representation

c-τ_0	$(x_0, 0)$ $(x_1, 0)$ $(x_1, 1)$ $(x_2, 0)$ $(x_2, 1)$
c-τ_1	$(x_0, 1)$ $(x_1, 2)$ $(x_2, 0)$ $(x_2, 1)$ $(x_2, 2)$

(b) Literal set

Figure 3.2: A c-table constraint.

It is NP-hard to generate a c-table including the minimum number of c-tuples [KW07]. Then minimizing a DNF can also be regarded as minimizing a c-table over Boolean variables, where DNF minimization is a well-known NP-complete problem [AHM+08]. In addition, c-tables can be constructed with different methods and heuristics. For example, we can construct c-tuples from decision trees [KW07] and also MDDs [XY13].

3.2.3 The MVD and MDD constraints

In addition to table representations, decision diagrams can also be used to define ad-hoc constraints, where the tuples in a constraint relation are represented as the paths in a decision diagram. Decision diagram constraints can be exponentially smaller than table constraints. In this subsection, we introduce the MVD and MDD constraints, where MVD is the non-deterministic MDD.

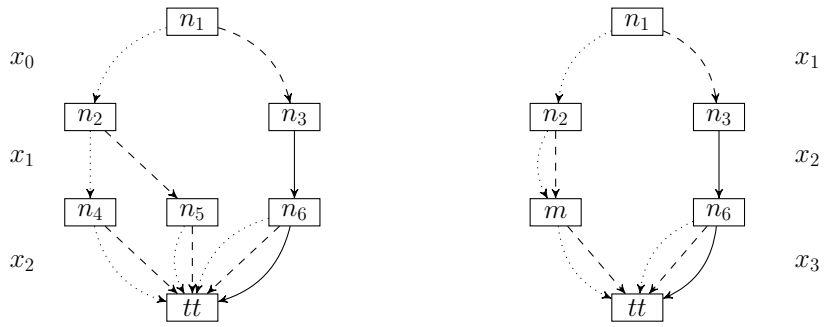
Definition 3.1. *An* Ordered Multi-valued Variable Diagram (MVD) *[AFNP14, VLS18, VLS19] with respect to an order O over r variables (O_i denotes the i-th variable) is a labelled layered directed acyclic graph denoted by the triple, $(\cup_{i=1}^{r+1} L_i, \cup_{i=1}^{r} E_i, label)$, where:*

- *L_i is a set of nodes at the i^{th} layer. L_1 only includes the root node and L_{r+1} the terminal node;*
- *E_i is a set of directed edges e pointing from a node $out(e) \in L_i$ to another node $in(e) \in L_{i+1}$, while $label(e)$ maps edge e to a value in $\mathcal{D}(O_i)$;*
- *for each node (edge), there is at least one path from root to the terminal node including the node (edge).*

MDD constraint [CY08, CY10] is a special case of MVD constraints, which is one of the most commonly used ad-hoc constraints. MDD constraints can be used to model various constraints, such as table constraints, regular constraints [Pes04], among constraints and sequence constraints [HPRS06]. Furthermore, various operations between MDD constraints can be done in polynomial time, e.g., the negation, intersection and union operations (more details can be found in [PR15]).

Definition 3.2. *An* Ordered Multi-valued Decision Diagram (MDD) *[SHMB90] with respect to an order O is a MVD $(\cup_{i=1}^{r+1} L_i, \cup_{i=1}^{r} E_i, label)$ w.r.t. O such that there is at most 1 edge $e \in E_i$ such that $out(e) = n_j$ and $label(e) = l$ for each value $l \in \mathcal{D}(O_i)$ and node $n_j \in L_i$.*

We use $mdd(c^*, O)$ to denote a MDD representing a constraint c^* w.r.t. an order O over $scp(c^*)$, where every tuple τ in $rel(c^*)$ corresponds to a path from root to the terminal in $mdd(c^*, O)$. Moreover, MDDs can be reduced by merging distinct nodes, where two nodes in the same layer are *distinct* if for each edge from one of two nodes, there is an edge from the other node such that two edges have the same label and point to the same node in next layer. In addition, we use the pReduce [PR15] algorithm to merge all distinct nodes in a MDD. However, we do not remove any redundant nodes from a MDD, where a node n_j in L_i is *redundant* if there are $|\mathcal{D}(O_i)|$ edges pointing from n_j to the same node. We remark that other MDD GAC propagators, such as MDD4R [PR14] and Compact-MDD [VLS18], also do not require removing redundant nodes. All MDDs considered in this thesis are ordered, and all edges in a MDD are directed edges.



(a) unreduced **(b)** reduced

Figure 3.3: Two MDD constraints where the dotted, dashed and solid lines respectively denote the value 0, 1 and 2.

Example 3.3. *Figure 3.3 shows two different MDD constraints to model the table constraint given in Figure 3.1. Each tuple in the constraint relation corresponds to a path from the root to the terminal node. The nodes n_4 and n_5 in the MDD given in Figure 3.3a are distinct, since the edges from n_4 are the same as those from n_5. After merging n_4 and n_5 as a new node m, we can get a smaller MDD shown in Figure 3.3b.*

3.2.4 The NFA and regular constraints

A *non-deterministic finite state automaton (NFA)* is a quintuple $(Q, \sum, \Delta, q_0, Q_t)$ consisting of a finite set Q of states and a finite set $\sum$ of input symbols and a transition function $\Delta : Q \times \sum \rightarrow 2^Q$ and an initial state $q_0 \in Q$ and a set $Q_t \subseteq Q$ of accepting states, where there is a transition in the NFA from a state $q_i \in Q$ to a state $q_j \in Q$ via a symbol $a \in \sum$ if $q_i \in \Delta(q_j, a)$. A string $a_1...a_r$ is accepted by the NFA if there is a sequence of states, $s_0, s_1, ..., s_r$, such that: $s_0 = q_0$, $s_i \in Q$ and $s_{i+1} \in \Delta(s_i, a_{i+1})$ for $0 \leq i < r$, and $s_r \in Q_t$. A NFA constraint c is a pair (G, O) such that O is an ordering over $scp(c)$, G is a NFA and $rel(c)$ is the set of tuples $\{(O_1, a_1), ..., (O_r, a_r)\}$ over $scp(c)$ such that the string $a_1...a_r$ is accepted by the NFA [QW06, CXY12].

The *regular constraint* is defined as a deterministic finite state automaton (DFA), where each DFA is a NFA $(Q, \sum, \Delta, q_0, Q_t)$ such that for each state $q_i \in Q$ and a symbol $a \in \sum$, the size $|\Delta(q_j, a)|$ is at most 1. Therefore, the regular constraint is a special case of the NFA constraint. NFA constraints can be decomposed into ternary table constraints and handled by use of table GAC propagators [QW06].

3.3 History of table AC/GAC propagators

This section surveys techniques and algorithms for GAC and AC on table constraints. There has been a large body of work on AC/GAC algorithms of table constraints. Table 3.1 gives an overview of the algorithms discussed in this section. Our goal is to explain key ideas which have been historically important as well as review more recent algorithmic ideas over the past decades. We will attempt to explain how the development of improvements in table AC/GAC algorithms has evolved and the key ideas. We review how algorithms have dealt with binary to general constraints and how they have exploited structure.

Techniques	Algorithms
GAC-scheme based	GAC-scheme, GAC-V, GAC-A, GAC-nextIn GAC-VA, GAC-nextDiff, Trie
Index	GAC-nextIn, GAC-VA, GAC-nextDiff, Trie, **AC5TC**
STR based	**STR, STR2, STR2+, STR3, STR-neg** **STR2w, STRbit, CT, CT-neg**
Residue support	AC7, GAC-scheme based, (G)AC3.1/AC2001, AC3.2, AC3.3 AC3rm, **AC3^{bit+r}, STR3, AC5TC, STR2w, STRbit, CT**
Bitwise Representation	**AC3^{bit},AC3^{bit+r}, STRbit, CT, CT-neg**
Incrementality	AC4, GAC4, AC5, AC6, AC7, GAC-scheme based, (G)AC3.1/AC2001 AC3.2, AC3.3, **STR based, GAC4R, AC5TC**
Reset	**GAC4R, CT**
Watched tuple	AC6, AC7, GAC-scheme, GAC-nextIn GAC-nextDiff, Trie, **STR3,STR2w**

Table 3.1: Overview table GAC/AC algorithms and techniques. The algorithms are sorted by year. The algorithms newer than STR [Ull07] are marked in **bold**.

3.3.1 AC algorithms

There has been considerable research in AC algorithms since the pioneering work of the AC3 algorithm [Mac77a] in 1977. AC3 enforces AC at the granularity of a single constraint, i.e., *coarse-grained*, and propagates domain changes from variables to other constraints whose scope includes those variables. From the perspective of binary CSPs, a constraint is an edge in the constraint graph (primal graph), thus, a coarse-grained AC algorithm works on edges. A coarse-grained approach for an AC can be attractive as the algorithm can be simple with low overheads because of simple data structures. In contrast to AC3, AC4 [MH86] works by enforcing AC at the granularity of a single variable's domain value, i.e., *fine-grained*, and was the first optimal AC algorithm in terms of time complexity. It tries to perform minimum work to maintain AC when a value is removed from a variable's domain. The trade-off compared to the coarse-grained AC3 is that AC4 maintains more fine-grained information including a list of supports and counters for the number of supports of each variable-value pair. In practice, AC4 was found to be outperformed by AC3 even though AC3 is not an

optimal algorithm [Wal93] which highlights that it is not sufficient to consider worst case complexity for efficient and practical constraint solving. Meanwhile, AC3 and AC4 can be regarded as special cases of AC5 [VHDT92]. AC6 [BC93, BC94b] is also optimal but reduces the space of AC4 by a factor of d (variable's domain size). AC7 [BFR99] extends AC6 by exploiting bidirectionality, i.e., both values v_i and v_j of a valid support (v_i, v_j) can simultaneously support each other.

In 2001, more than two decades after AC3, an optimal coarse-grained algorithm called AC3.1/AC2001 [BR01, BRYZ05, ZY01] was found, which outperforms AC3. AC3.1/AC2001 gained efficiency with *residues*, where residues are the supports found previously and saved for later use. Ordering supports and using residues amortizes support checking, reducing it by a factor of d making AC3.1/AC2001 optimal. AC3.2 and AC3.3 [LBH03] extend AC3.1/AC2001 by partially and fully involving the bidirectionality of AC7 respectively. AC3^{bit} [LV08] is the first practical AC algorithm exploiting bitwise operations. Essentially it uses bit vectors to represent the domain and supports. The speed advantage is due to the $(O(1))$ operations available on bit vectors giving speedups due to the word size. AC3^{bit+r} [LV08] further extends AC3^{bit} to residues, and is still a state-of-the-art AC algorithm.

We highlight that many AC algorithms can be extended to GAC. For example, AC3, AC7, AC3.1/2001 and AC3^{rm} [LH07] are respectively extended to the algorithms GAC3 [Mac77b], GAC-scheme [BR97], GAC3.1/2001 [ZY01, BRYZ05], and GAC3^{rm} [LH07] which can be used to enforce GAC on any constraints including the table and MDD constraints. The older AC algorithms also contribute to the development of modern GAC algorithms, e.g., design choices such as propagation granularity, and optimizations like residue, bitwise representation, and multi-directionality.

3.3.2 GAC algorithms

We now review the GAC algorithms for table constraints categorizing them by algorithm design choices, leveraging from optimization techniques to constraint representations.

GAC schema

GAC-scheme [BR97] is a framework enforcing GAC on any constraint by implementing a *seekingSupports* function, it searches supports for each domain value. If *seekingSupport* finds a support for a value, the value is GAC; otherwise the value is inconsistent. GAC-V and GAC-A are the two approaches following the GAC-scheme but with different ways of seeking supports. GAC-V iterates over valid tuples until one satisfying the constraint, while GAC-A iterates over the allowed tuples of the constraint until a valid one is found, where allowed tuples denote the tuples included in the constraint relation. In GAC-VA, it alternates traversal of the list of valid and allowed tuples. GAC-VA was shown to be more

robust than GAC-A or GAC-V when constraint tightness is close to both extremes, i.e., close to high end 1.0 or low end 0.0. GAC-nextln [GJMN07], GAC-nextDiff and Trie [LR05] are also based on the GAC-scheme, but use indexing.

Indexing

Indexing is a classical way of optimizing data search by an efficient data structure. It has also been used in GAC algorithms. GAC-nextln builds an index over a constraint c such that for each literal (x, a), the tuple of c containing (x, a) links to the following tuple also containing (x, a). GAC-nextDiff also builds indexes between tuples but links to the following tuple having different literals for the same variable. Another indexing data structure is a trie [LR05] such that converting tables into trie can speed up the process of seeking supports. The AC5TC [MVHD12] algorithm and its variants also benefit from indexing on top of the generic AC5 algorithm. Experiments suggest that AC5TC variants can be more efficient than STR2+, STR3 and MDDc under low constraint arity but slower under high arity [LLY15b].

Residue Support

The idea of *residue support* is to record previously found supports, called residues, then seek supports from the residues to skip some checkings. It is a technique from the AC algorithms, e.g., AC3.1/AC2001, which uses residues to get optimality and efficiency. Additionally, following the AC algorithms AC3.2, AC3.3, and AC3rm [LBH03, LH07], residues can also work in a multi-directional way, i.e., when a support is found, it can be used as a residue for all values occurring in the support besides the value for which it was seeking support. Many newer table GAC algorithms use the residue idea, e.g., STRbit [WXYL16] and CT [DHL+16]. We will review these two algorithms in the following subsections.

Simple Tabular Reduction

Simple Tabular Reduction (STR) [Ull07] is one of the most successful techniques for filtering table constraints. At least 15 STR based algorithms were proposed for table constraints. The idea of STR is to remove invalid tuples from tables (shrinking the tables dynamically) as search goes deeper, and restore them upon backtracking. STR reduces the number of tuples of a table as the search goes deeper, saving unnecessary tuple checks. Experiments showed that STR can be very effective in shrinking tables—the average reduced table size can be smaller than the original table by 1-3 orders of magnitude [LLY12].

STR2 [Lec08, Lec11] optimises the basic STR algorithm in two ways. First, the variables whose domain has not changed since the last invocation are skipped when checking the validity of a tuple. Second, support checking is skipped when all domain values are consistent. STR2+ further adds a data structure to maintain the domain size of a variable at the last

time a particular constraint is processed during the search. Experiments in [LLY15b] show that STR2+ is about 20% faster and 2X faster than STR when the arity of constraints are 10 and 16 respectively. Then the STR2w [LLY15a] algorithm makes STR2 more efficient by using the *watched tuple* techniques.

STR3 [LLY12, LLY15b] is a STR algorithm which uses a different representation, called dual table. In the dual table, each variable-value pair is mapped to a set of tuples containing the pair. Figure 3.4a shows an example of the dual table, which is equivalent to the table constraint in Figure 3.1. For each table constraint and variable value, the supports of the variable value are recorded in the dual table, e.g., $\{\tau_0, \tau_1, \tau_2, \tau_3\}$ are the supports of $(x_0, 0)$. This illustrates a different change of representation in the data structures used to represent the table (AC4 also uses a fine-grained data structure). Unlike STR2, this makes STR3 a fine-grained algorithm and it is designed to maintain GAC on the dual table during search, making it incremental. STR3 has the property of being *path-optimal*, i.e., each element of a table is traversed at most once along a search path. Their experiments show that the dynamic table size has the most significant factor affecting the performance of STR3, i.e., STR3 is faster than STR2 when tables remain large (table reduction is less effective) while STR3 is slower than STR2 if the table size becomes small. The state-of-the-art GAC algorithms STRbit [WXYL16] and CT [VLS17] also use STR ideas among others. In addition, STR is also extended to handle negative tables, such as STR-neg [LLGL13] and CT-neg [VLS17].

x_0		x_1		x_2	
0	$\{\tau_0, \tau_1, \tau_2, \tau_3\}$	0	$\{\tau_0, \tau_1\}$	0	$\{\tau_0, \tau_2, \tau_4\}$
1	$\{\tau_4, \tau_5, \tau_6\}$	1	$\{\tau_2, \tau_3\}$	1	$\{\tau_1, \tau_3, \tau_5\}$
		2	$\{\tau_4, \tau_5, \tau_6\}$	2	$\{\tau_6\}$

(a) The table representation for STR3.

x_0		x_1		x_2	
0	$\{(\theta_0, 1111)\}$	0	$\{(\theta_0, 1100)\}$	0	$\{(\theta_0, 1010), (\theta_1, 1000)\}$
1	$\{(\theta_1, 1110)\}$	1	$\{(\theta_0, 0011)\}$	1	$\{(\theta_0, 0101), (\theta_1, 0100)\}$
		2	$\{(\theta_1, 1110)\}$	2	$\{(\theta_1, 0010)\}$

(b) The table Representation for STRbit, where θ_0 is the index of the word for τ_0 to τ_3 and θ_1 is for τ_4 to τ_6.

Figure 3.4: Table Representations for STR3 and STRbit.

Bitwise Representation

Wang et al. [WXYL16] proposed a bitwise encoding of the dual table representations together with the algorithms STRbit and STRbit-C. To get the bitwise representation, the original table is first partitioned so that each sub-table has w tuples where w corresponds to the natural word size of a processor with $O(1)$ bit vector operations. Figure 3.4b gives an example of the bitwise representations of Figure 3.4a. In this example, we assume $w = 4$ and partition the table into two parts with θ_0 and θ_1 as the index. For each variable value

and sub-table, a word in a bit vector is used to record whether the tuples in the sub-table are supports of the variable value, i.e.,, the i^{th} bit in the word is 1 if the i^{th} tuple in the sub-table is a support, otherwise the i^{th} bit is 0. If there is no support from the sub-table, the word equals 0 and can be skipped. Experiments showed that STRbit was one of the state-of-art algorithms, being up to 25X faster than STR3 and 70X faster than STR2.

Compact-table (CT) is another state-of-the-art algorithm, also based on bitwise representation. Both CT and STRbit use bit vectors (sparse sets [BT93]) to record all valid tuples in a table (non-zero words in the bit vectors) during search. The difference between CT and STRbit is that for each table constraint, CT does not skip the zero words in the bit vectors recording supports of variable values, thus, those bit vectors have the same number of words and can share the sparse set which records non-zero words in the bit vector recording valid tuples. In addition, the (STRbit) CT algorithm also records the last found support for each variable value, which is similar to (AC2001) AC3^r [LZBF04].

Incrementality

For the most incrementality techniques, some additional data structures are maintained during search, in order to avoid repeatedly accessing invalid supports (tuples). A motivation is that if a support of a variable value is invalid, then it is still invalid after shrinking some variable domains. Incrementality techniques appear in many AC/GAC algorithms: (i)For the GAC-scheme based, GAC2001/3.1/3.2/3.3, STR3 and STRbit algorithms, the last valid support of each variable value is maintained, thus, we only need to consider the supports before the last support when the last support becomes invalid; (ii) In the STR based algorithms, current tables, i.e.,, invalid tuples are removed, are maintained by using sparse (bit-)sets; (iii) In the GAC4 algorithm [MM88], the valid supports of each variable value are maintained by using linked lists or sparse sets;

Reset

The *reset* technique is introduced in GAC4R [PR14], and used in many recent GAC algorithms, such as STRbit, CT and smartCT. The idea is to choose whether to incrementally maintain deletions or to rebuild the data structures from scratch.

Watched Tuple

The *watched tuple* technique, inspired by SAT solvers, is to associate each watched tuple with a list of values that depend on this tuple as a proof of support. This simplifies checking whether a support is no longer correct. Should a tuple be removed, its watchers must be reattached, if possible, to another valid tuple. The algorithms applying such technique include GAC-shceme, GAC-nexIn, GAC-nextDiff, Trie, STR3, STR3-C and STR2w.

3.4 History of GAC propagators for non-ordinary tables

Many non-ordinary tables (shown in Figure 3.5) have been proposed to improve the expressive power of table constraints. The compressed table (c-table) [KW07] work on the Cartesian Product representation (c-tuple) of tuples. The sequence table [Rég11b] encodes a table as a set of tuple sequences, where each tuple sequence is defined by a minimum tuple and a maximum tuple w.r.t. an ordering. The short table [JN13] compresses table constraints by hiding the variables whose domain values are always supported. The sliced table [GHLR14] compresses tables by first grouping tuples of a table (slicing) and then decomposing each group (a sub-table) into two tables with a smaller arity where the original sub-table is obtained from the join of the two smaller tables. The smart table [MDL15] (basic smart table [VLDS17]) partitions a table into subsets of tuples where each subset is modelled as an acyclic CSP using specific unary and binary constraints (unary constraints). The segmented table [ALM20] encodes a subset of tuples as a set S of table constraints such that $scp(c_1) \cap scp(c_2) = \emptyset$ for any 2 different constraints c_1, c_2 in S.

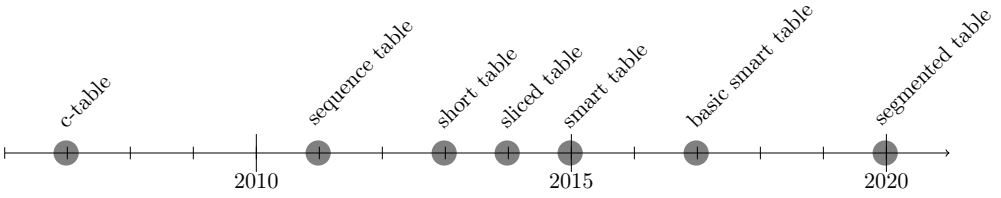


Figure 3.5: History of non-ordinary tables.

Various table GAC propagators can be directly extended to handle non-ordinary tables. [KW07, Rég11b] show that the GAC-scheme algorithm can handle the c-table and sequence table. Then the STR algorithm is extended to the non-ordinary table algorithms shortSTR2 [JN13], STR2-C, STR3-C [XY13], STR-slice [GHLR14], STRbit-C [WXYL16], smartSTR [MDL15] and seg [ALM20] (the "-C" refers to a c-tuple version of the algorithm). In addition, the CT algorithm can also be used to handle the c-table and basic smart table [VLDS17]. Such algorithms are competitive when the non-ordinary tables have enough compression ratio. For example, Xia and Yap [XY13] show that STR2-C is faster than STR2 when the Cartesian product compresses tables by more than 75%, in addition, Wang et al. [WXYL16] also show that STRbit-C is faster than STRbit when the compression ratio of c-table to table is more than 3.

Although non-ordinary tables can be much more compact than table for some extreme cases, the experimental results given in [VLS19] show that the state-of-the-art GAC propagator of table constraint still outperforms that of the non-ordinary tables on a large set of benchmarks. It still lacks of a non-ordinary table GAC propagator which can overall outperform table GAC propagators on the "standard" benchmarks.

3.5 History of MDD GAC propagators

Besides the table-based representation, we can convert any table constraint into an equivalent MDD constraint [CY10]. In the best case, the MDD constraint can get exponential compression. Like a tree/trie, a MDD can get compression due to sharing in the tree part of the graph but it also gets sharing of children. In practice, it was observed that the more compact the MDD, the faster the MDD based filtering algorithms become. In an analog to table support, a value is GAC if there exists a path of valid domain values from the root node to the terminal node.

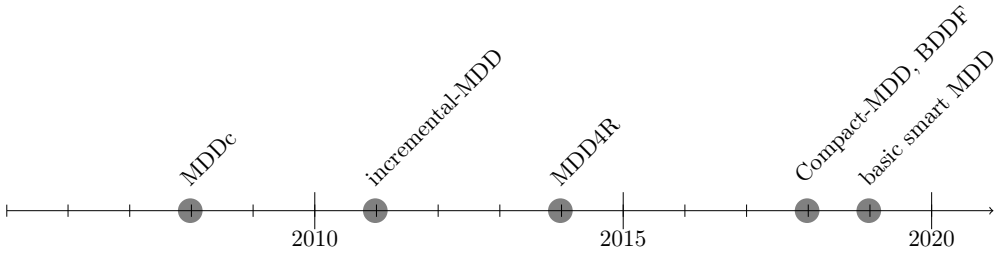


Figure 3.6: History of MDD GAC propagators.

A number of MDD GAC algorithms have been proposed (see Figure 3.6), including MDDc [CY10], incremental-MDD [GSS11], MDD4 [PR14], BDDF [VP18], Compact-MDD (CD) [VLS18] and basic smart MDD [VLS19]. The MDDc is the first GAC algorithm for MDD constraints. It recursively traverses the MDD in a top-bottom manner and enforces GAC during the process. At each MDD node, MDDc records the consistency of the node, i.e., whether the node can reach a true terminal through a path whose values are all present in current variable domains. This guarantees that MDDc explores each MDD node at most once. Another optimization technique is the *early-cutoff*, which remembers the level at and below of the MDD where all values in the domains of variables are consistent. As a result, MDDc can skip the unexplored sub-parts of any consistent node at or below the level. The MDD data structure also serves as a natural index. MDD4 [PR14] adds incrementality by maintaining the validity of MDD edges kept as the supports of domain values. The *reset* technique is introduced giving GAC4R and MDD4R. The idea is to choose whether to incrementally maintain deletions or rebuild the data structures from scratch.

Although MDD constraints can be more compact than table constraints, experimental results in [WXYL16, DHL+16] show that the state-of-the-art table GAC algorithms using bitwise operations, such as STRbit and CT, overall outperform the MDD GAC algorithms on a large set of benchmarks. In order to reduce the large gap between the compression ratio of MDD constraints and the efficiency of MDD GAC propagators, the Compact-MDD (CD) algorithm [VLS18] extends the principles of CT to handle MDD constraints. At the same time, the CD algorithm is also improved by compressing MDD constraints with the

semi-MDD [VLS18] and basic smart MDDs [VLS19]. CD and its variants can reduce the gap. However, the experimental results given in their papers [VLS18, VLS19] also show that the CT algorithm is still faster than MDD GAC algorithms on a large set of benchmarks.

3.6 Conclusion

Ad-hoc constraints are very useful for modelling CSPs. For example, ad-hoc constraints can be used to model the specific constraints which do not fit well with special purpose global constraints. The table and MDD constraints are two well-known ad-hoc constraints which have been implemented in various CSP solvers. Many efficient algorithms can be used to enforce GAC on the table and MDD constraints. In addition, a large number of non-ordinary table constraints, such as the c-table, short table and smart table constraints, are proposed to improve the succinctness of table constraints.

In this chapter, we survey the GAC propagators of the table, non-ordinary table and MDD constraints. A large body of techniques have been proposed to improve the GAC propagators, e.g., the bitwise representation is one of the key techniques used in the state-of-the-art table GAC propagators. Although non-ordinary tables and MDDs can be much smaller than tables in some cases, the state-of-the-art GAC propagators of table constraints can still outperform those of the non-ordinary table and MDD constraints on a large set of benchmarks. In addition, the propagators proposed in the past 5 years, such as the STRbit and CT algorithms, are still state-of-the-art table GAC propagators. Therefore, we believe that it is time to investigate ad-hoc constraint GAC propagators from different perspectives. We will try to encode ad-hoc constraints into binary constraints, and then use AC propagators to enforce GAC on the constraints.

Chapter 4

Binary encodings

4.1 Introduction

Looking at the history of CSPs (Constraint Satisfaction Problems), we can see that binary constraints play an important role in the early stage of studying CSPs. Before 1990, many papers show that binary constraints can be applied to model a wide range of problems, such as the picture processing problems [Mon74], graph problems [McG79] and temporal reasoning problems [All83, VK86, Tsa87]. In addition, a large body of algorithms were designed to solve binary CSPs, such as the backtracking search [Gas77] algorithm, the AC3 [Mac77a] propagation algorithm, the intelligent backtracking search algorithm [SS77, Doy79] and the forward checking algorithm [HE80]. Moreover, a survey paper about CSPs published in 1992 [Kum92] only focuses on discussing the techniques used to solve binary constraints. Around that time, binary encodings, including the dual encoding [DP89] and hidden variable encoding [RPD90], were proposed to transform non-binary constraints into binary constraints in order to exploit many efficient algorithms of binary constraints.[1]

Between 1990 and 2000, a number of efficient GAC algorithms were proposed to directly enforce GAC on non-binary constraints rather than encoding them into binary constraints and using AC propagators. For example, the generic GAC propagators which can be used for handling arbitrary non-binary constraints [BR97], and the specific GAC propagators for special purpose global constraints, such as the AllDifferent constraint [Rég94], sequence constraint [BC94a] and Global Cardinality Constraint (GCC) [Rég96]. In particular, the solving algorithms using special purpose global constraints can be much faster than those using binary encodings. For example, GAC propagator on AllDifferent constraints can be more efficient than AC propagators on binary inequalities [Rég94, SW99a]. In addition, a wide range of real life problems can be naturally modelled with non-binary constraints. Therefore, non-binary constraints start to attract attention [Bes99].

After 2000, there are more algorithms [Rég11a] proposed to directly handle non-binary

[1] All non-binary constraint relations can be transformed into binary constraints by using an encoding with addition variables, i.e., hidden variables [Dec90b]. Additionally, various global constraints can also be decomposed into binary constraints [GSW00].

constraints. For example, many GAC propagators for ad-hoc constraints were proposed after 2000 (see Chapter 3). In addition, empirical experimental results show that GAC propagators of non-binary constraints can be more efficient than AC propagators [SW99a, SSW00], even for table constraints, e.g., the experimental results given in [Lec11] show that the STR2 algorithm can outperform AC propagators with various binary encodings. Improvements in GAC propagators have led to a folklore believe that AC propagators on binary encodings do not compete with GAC propagators on non-binary constraints. Since 2010 to before this work (2019), the most algorithms of solving CSPs directly deal with non-binary constraints.

In this thesis, we will show that specialized AC propagators with binary encodings can outperform the state-of-the-art GAC propagators on various non-binary ad-hoc constraints, e.g., the state-of-the-art GAC propagators of table and MDD constraints, which means that the folklore about AC propagators versus GAC propagators is misleading. It is not always the case that GAC propagators on non-binary constraints are more efficient than AC propagators with binary encodings. In the rest of this chapter, we introduce the well-known binary encodings which can be used to transform non-binary ad-hoc constraints into binary constraints. We remark that these binary encodings were proposed 20 years ago and mainly focus on handling table constraints.

4.2 Dual Encoding

In order to exploit the efficient algorithms of binary constraints to solve non-binary CSPs, such as the algorithms using tree structures [Fre82, DP87], Dechter and Pearl [DP89] proposed the first binary encoding called dual encoding to transform non-binary CSPs into binary constraints based on their dual graphs. The dual encoding regards every constraint c_i in a CSP as a hidden variable hv_i (which is also called c-variable in [DP89] and dual variable in [SW99b]) such that each tuple in $rel(c_i)$ is encoded as a value in $\mathcal{D}(hv_i)$, i.e., $\mathcal{D}(hv_i) = rel(c_i)$. Then for each two constraints c_i, c_j in the CSP such that $scp(c_i) \cap scp(c_j) \neq \emptyset$, the dual encoding uses a binary constraint c_{ij} between the hidden variables hv_i, hv_j to connect the values (tuples) in the hidden variable domains $\mathcal{D}(hv_i), \mathcal{D}(hv_j)$.

Definition 4.1 (dual encoding)**.** *The dual encoding of a CSP $P = (X, C)$ is a binary CSP (H, DC), where $H = \{hv_i | c_i \in C\}$ is a set of hidden variables hv_i such that the domain $\mathcal{D}(hv_i)$ is equal to $rel(c_i)$, and $DC = \{c_{ij} | scp(c_i) \cap scp(c_j) \neq \emptyset\}$ is a set of binary constraints c_{ij} such that $scp(c_{ij}) = \{hv_i, hv_j\}$ and $rel(c_{ij}) = \{\{(hv_i, \tau_1), (hv_j, \tau_2)\} \mid \tau_1 \in rel(c_i), \tau_2 \in rel(c_j), \tau_1[S] = \tau_2[S], S = scp(c_i) \cap scp(c_j)\}$.*

Enforce various consistencies on the dual encoding, such as Forward Checking [HE80], AC [Mac77a], Neighborhood Inverse Consistency [FE96] and Singleton Arc Consistency (SAC) [DB01], can achieve different consistencies on the original CSPs [JJNV89, BVB98,

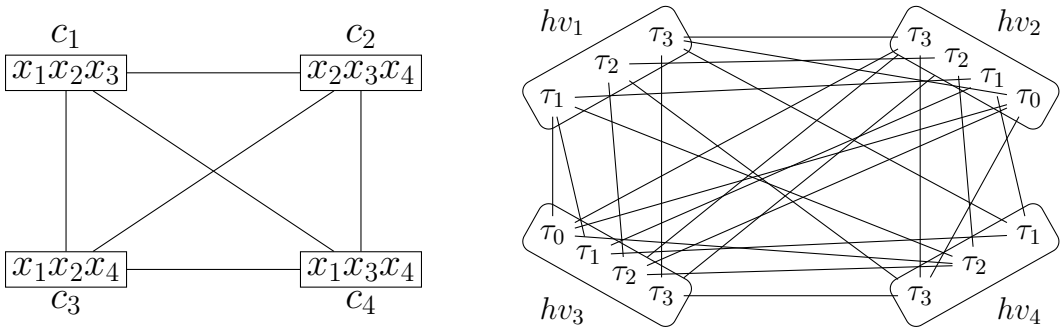
SW99b, BCVBW02, BSW08]. For example, enforce AC on the dual encoding can achieve PWC on the original constraints [JJNV89].

Example 4.1. *Let P be a CSP (X, C) such that $X = \{x_1, \ldots, x_4\}$, $C = \{c_1, c_2, c_3, c_4\}$,*

- *$\mathcal{D}(x_i) = \{0, 1\}$ for $x_i \in X$, $scp(c_1) = \{x_1, x_2, x_3\}$, $scp(c_2) = \{x_2, x_3, x_4\}$, $scp(c_3) = \{x_1, x_2, x_4\}$, $scp(c_4) = \{x_1, x_3, x_4\}$, and*
- *$rel(c_1)$, $rel(c_4)$ are denoted as $\{\tau_1, \tau_2, \tau_3\}$ w.r.t. $< x_1, x_2, x_3 >$ and $< x_1, x_3, x_4 >$, while $rel(c_2)$ and $rel(c_3)$ are denoted as $\{\tau_0, \tau_1, \tau_2, \tau_3\}$ w.r.t. $< x_2, x_3, x_4 >$ and $< x_1, x_3, x_4 >$,*

where $\tau_0 =< 0, 0, 0 >$, $\tau_1 =< 0, 0, 1 >$, $\tau_2 =< 0, 1, 0 >$, $\tau_3 =< 1, 0, 0 >$, and for any $R \subseteq \{\tau_0, \cdots, \tau_4\}$, the set of tuples $\{\{(v_1, a_1), \cdots, (v_r, a_r)\} | < a_1, \cdots, a_r > \in R\}$ is denoted as R w.r.t. variables $< v_1, \cdots, v_r >$.

Figure 4.1a shows the dual graph of P, each node in the graph denotes a constraint. Then Figure 4.1b is the dual encoding of P. Every rectangle node in the figure denotes a hidden variable and the set of values in the rectangle node denotes the domain of the hidden variable. In addition, each edge in the figure connecting 2 values τ, τ' of two hidden variables hv_i, hv_j denotes a tuple $\{(hv_i, \tau), (hv_j, \tau')\}$ in the constraint relation $rel(c_{ij})$, e.g., $rel(c_{13})$ includes 4 tuples $\{\{(hv_1, \tau_1), (hv_2, \tau_0)\}, \{(hv_1, \tau_1), (hv_2, \tau_1)\}, \{(hv_1, \tau_2), (hv_2, \tau_2)\}, \{(hv_1, \tau_3), (hv_2, \tau_3)\}\}$.



(a) Dual graph **(b)** Dual encoding

Figure 4.1: An example of the dual encoding.

It is impractical to use existing AC algorithms, such as the AC3^{bit} algorithm, to directly enforce AC on the dual encoding, since the dual encoding runs out of memory on many CSP benchmarks (see the experimental results shown in [Lec11]). Then a specialized AC propagation algorithm called PW-AC [SS05] was proposed to enforce AC on the dual encoding. In addition, the PW-AC algorithm can also be optimized with bitwise representations [SC18]. However, PW-AC still runs out of memory on many CSP benchmarks, and it is slower than the state-of-the-art table GAC algorithms, such as the CT and STRbit algorithms, on a large set of benchmarks (see the experimental results given in [SC18]).

4.3 Hidden Variable Encoding

Inspired by the dual encoding, Ross et al. [RPD90] introduced the second binary encoding called Hidden Variable Encoding (HVE) to transform non-binary CSPs into binary constraints by adding hidden variables while generalizing the notion of equivalence. HVE encodes every constraint c_i in a CSP as a hidden variable hv_i such that every value in the hidden variable domain $\mathcal{D}(hv_i)$ corresponds to a tuple in the constraint relation $rel(c_i)$. At the same time, it uses binary constraints to connect the tuples in the domain $\mathcal{D}(hv_i)$ with the values in the domains of the variables in $scp(c_i)$.

Definition 4.2 (HVE)**.** *The* hidden variable encoding (HVE) *of a CSP $P = (X, C)$ is a binary CSP $(X \cup H, HC)$, where $H = \{hv_i | c_i \in C\}$ is a set of hidden variables hv_i such that the domain $\mathcal{D}(hv_i)$ is equal to $rel(c_i)$, and $HC = \{c_i^x | x \in scp(c_i), c_i \in C\}$ is a set of binary constraints c_i^x such that $scp(c_i^x) = \{x, hv_i\}$ and $rel(c_i^x) = \{\{(x, a), (hv_i, \tau)\} | a \in \mathcal{D}(x), \tau \in rel(c_i), (x, a) \in \tau\}$. We call the variables in X as original variables.*

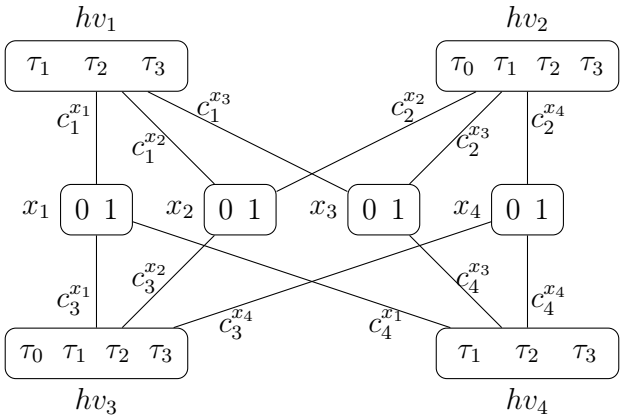


Figure 4.2: An example of the hidden variable encoding.

Example 4.2. *Figure 4.2 gives the HVE encoding of the CSP P shown in Figure 4.1b. Every rectangle node in the figure denotes a variable, and the set of the values shown in the node denotes the variable domain, e.g., $\{\tau_1, \tau_2, \tau_3\}$ is the variable domain $\mathcal{D}(hv_1)$, while each edge corresponds to a binary constraint $c_i^{x_j}$ between the variables hv_i and x_j, where $rel(c_i^{x_j})$ can be directly constructed based on Definition 4.2, e.g., $rel(c_1^{x_1}) = \{\{(x_1, 0), (hv_1\tau_1)\}, \{(x_1, 0), (hv_1\tau_2)\}, \{(x_1, 1), (hv_1\tau_3)\}\}$.*

Many local consistencies on the HVE encoding are equal to GAC on the original CSPs [BCVBW02, BSW08]. For example, enforcing AC, Neighborhood Inverse Consistency, Path Inverse Consistency [FE96], Restricted Path Consistency [Ber95] and max Restricted Path Consistency [DB97] on the HVE encoding achieves GAC on the original CSPs. The strength of these consistencies on the dual encoding can be stronger than that of GAC on the original CSPs. Correspondingly these consistencies on the dual encoding can be stronger than that

on the HVE encoding. Two main advantages of the HVE encoding are: (i) the backtracking search algorithm on HVE instances can have the same search tree as that on the original CSPs, and (ii) the HVE encoding can be much more compact than the dual encoding.

Existing AC algorithms can be directly used to enforce AC on the HVE encoding. In addition, we can also design specialized AC algorithms exploiting the special structure of the HVE encoding, i.e., the constraint graph of the binary constraints encoding a non-binary constraint is a star graph. Mamoulis and Stergiou [MS01] introduced a specialized AC algorithm called HAC to enforce AC on the HVE encoding by emulating the GAC2001 algorithm. The experimental results given in [SS05] show that the HAC algorithm can be competitive with the GAC2001 algorithm. However, it has been shown in [Lec11] that many GAC algorithms of table constraints, such as the STR2 and MDDc algorithms, can overall outperform the HAC algorithm and the state-of-the-art AC propagators $AC3^{bit}$ on the HVE encoding (see the experimental results given in [Lec11]).

4.4 Double encoding

The dual encoding and the HVE encoding have advantages in different ways. For example, enforcing AC on the dual and HVE encoding can respectively achieve PWC and GAC on the original constraints. In addition, the HVE encoding includes both the original variables and hidden variables, therefore, the backtracking search algorithm can branch on two kinds of variables for the HVE encoding. In order to exploit the advantages of both the encodings, Stergiou and Walsh [SW99b] introduced a binary encoding called double encoding to combine the dual encoding and the HVE encoding.

Definition 4.3 (double encoding)**.** *The* double encoding *of a CSP $P = (X, C)$ is a binary CSP $(X \cup H, HC \cup DC)$ such that $(X \cup H, HC)$ is the HVE encoding of P and (H, DC) is the dual encoding of P, where X is the original variables and H is the hidden variables.*

Enforcing AC on the double encoding can achieve FPWC (both GAC and PWC) on the original CSPs. Hence, the strength of AC on the double encoding can be stronger than that on the dual and HVE encodings. Additionally, the double encoding includes both the original variables and hidden variables, thus, the backtracking search algorithm can also branch on two kinds of variables. However, the double encoding requires more space than the dual encoding, and running out of memory on many benchmarks (see the experimental results shown in [Lec11]). Therefore, it is impractical to directly enforce AC on the double encoding using existing AC algorithms for many benchmarks.

4.5 Conclusion

Although binary constraints play an important role in the early stage of studying CSPs, non-binary constraints become more and more important, since many real life problems can be naturally modelled with non-binary constraints, and GAC propagators of many non-binary constraints can be more efficient than AC propagators with binary encodings. Most research about binary encodings (over the past decade and more) has been on the former as it is believed that binary encoding is not practical. The folklore on binary encodings versus non-binary constraints is that GAC algorithms of non-binary constraints are more efficient than using the AC propagators on binary encodings of the original non-binary model.

We observe that there has been little work on studying new binary encodings and specialized AC propagators for binary encodings. The state-of-the-art binary encodings are still the well known encodings, i.e., the dual encoding and HVE, which were proposed in 30 years ago. Moreover, there are only a few specialized AC propagators, i.e., the HAC and PW-AC algorithms, which exploit special properties of binary encodings. We believe that if we can give better binary encodings and more efficient specialized AC propagation algorithms for the binary encodings, then AC algorithms with the binary encodings can be more efficient than GAC algorithms on various non-binary constraints.

In this thesis, we will show that the folklore is misleading. We will give new binary encodings to transform non-binary ad-hoc constraints into binary constraints, and show that specialized AC propagation algorithms with new binary encodings can significantly outperform the state-of-the-art GAC algorithms of the table and MDD constraints.

Part B

Table Constraints

Chapter 5

Arc Consistency revisited

5.1 Introduction

Binary constraint is a general representation of constraints and is used in Constraint Satisfaction Problems (CSPs) to model/solve discrete combinatorial problems. Historically, work on constraint satisfaction began with binary CSPs, problems with at most two variables per constraint, and many algorithms have been proposed to maintain Arc Consistency (AC) on binary constraints. In addition, several well-known binary encoding methods can be used to transform non-binary constraints into binary constraints. The seminal work of Mackworth [Mac77a] proposed a basic local consistency, arc consistency, which has been the main reasoning technique used in constraint solvers for CSPs. However, many problems are more naturally modelled with non-binary constraints. Furthermore, non-binary constraints can be directly handled with Generalized Arc Consistency (GAC) propagators. In more recent times, research has focused on non-binary constraints and efficient GAC propagators using clever algorithms, representations and data structures. Although several well-known binary encoding methods can be used to transform non-binary constraints to binary constraints, improvements in GAC algorithms have led to a "folklore belief" that AC algorithms on the binary encoding of non-binary constraints do not compete with GAC algorithms. The folklore has also spurred major developments in GAC algorithms.

In this chapter, we first show with experimental comparisons of binary encodings with state-of-the-art table GAC algorithms the reasons why binary encoding with existing AC algorithms are outperformed by GAC on non-binary table constraints. This explains the basis of the folklore belief that AC on binary encodings is not competitive. However, we then show this belief to be flawed. We propose a new algorithm to enforce AC on binary encoded instances, which address 2 factors: (i) a more efficient propagator for binary constraints; and (ii) control the interaction between the search heuristic and the binary encoded model. Finally, preliminary experiments show our AC algorithm on binary encoded instances can be competitive with state-of-the-art table GAC algorithms on the original non-binary table constraints, such as CT[DHL+16] and STRbit [WXYL16]. This result is surprising and is contrary to the "folklore" on AC vs GAC algorithms.

5.2 History and problems

In this section, we revisit the question whether non-binary constraints are better handled directly using non-binary constraint propagators or the non-binary constraints are encoded to binary constraints and processed by binary constraint propagators. We focus on the comparison between binary constraints and table constraints which are the most general representation for constraints in CSPs. We start with a chronology of binary encodings and corresponding consistency algorithms (if any). In 1998, [BVB98] showed on some instances, forward checking (FC) with backtracking search on binary encoded instances can be faster than solving the non-binary instances directly. In 1999, experimental results in [SW99b] showed that enforcing AC on the dual encoding instances is very expensive. In 2001, [MS01] proposed HAC showing that HAC is competitive with GAC, and GAC algorithms can be mapped to the corresponding AC algorithms on binary encoded instances. In 2005, [SS05] showed that binary encodings can be competitive with the non-binary representation. It also showed that a specialized algorithm, called PW-AC, can work well on dual encoding instances. HAC and PW-AC can be more efficient than GAC-2001 in some cases. However, they only tested some special cases for the dual encoding. In 2011, [Lec11] showed that the dual and double encodings run out of memory on many benchmarks, and STR2+ can outperform HAC and HVE+AC3^{bit+rm}.[1] In 2018, [SC18] showed that CT can overall outperform PW-AC (see Chapter 4 for more details about table constraints).

During the past decades, many AC and table GAC algorithms were proposed. AC algorithms check whether a variable value has a valid support on another variable domain, and many methods are proposed to reduce the cost of AC consistency algorithm during search. e.g., AC3[Mac77a], AC4[MH86], AC5[VHDT92], AC6[BC94b], AC7[BFR99], AC8[CJ98], AC2001[BRYZ05], AC3^{rm}[LV08], AC3^{bit+rm} and NAC4[DVH10], where a lot of AC algorithms focus on optimizing the *seekSupport* function. Some GAC algorithm also exploit the methods from AC algorithms, e.g., GAC4[MM88], GAC2001[BRYZ05] and AC5TC[MVHD12]. Over the period from 2007-2018, there has been considerable research efforts expanded on GAC algorithms, but there has been little work on new AC algorithms given the shift to GAC algorithms. Many of the GAC algorithms use special ideas to make GAC more efficient. An incomplete list is as follows: reducing the size of tables during search, e.g., algorithms using simple table reduction during search[Ull07, Lec11, LLY12], algorithms using decision diagrams [PR14, CY10, VLS18], algorithms using compressed tables [XY13, KW07, JN13, GHLR14], and bit set representations [WXYL16, DHL$^+$16, VLS17, VLDS17]. Some of the state-of-the-art GAC algorithms are CT[DHL$^+$16] and STRbit[WXYL16] which combine bit sets with simple table reduction

[1] (HVE+AC3^{bit+rm} means using the AC3^{bit+rm} on the HVE encoding).

Series	100%	80%	60%	40%	20%
Rand-5-12	50/50	50/50	50/50	50/50	50/50
Rand-15-23	25/25	25/25	25/25	25/25	25/25
Rand-10-60	50/50	50/50	50/50	50/50	50/50
Nonogram	88/170	78/170	71/170	53/170	40/170
Tsp	15/45	15/45	15/45	15/45	15/45
Jnh	50/50	40/50	25/50	2/50	0/50
Mdd-7-25	25/25	25/25	25/25	25/25	25/25
Rand-3-20	30/50	0/50	0/50	0/50	0/50

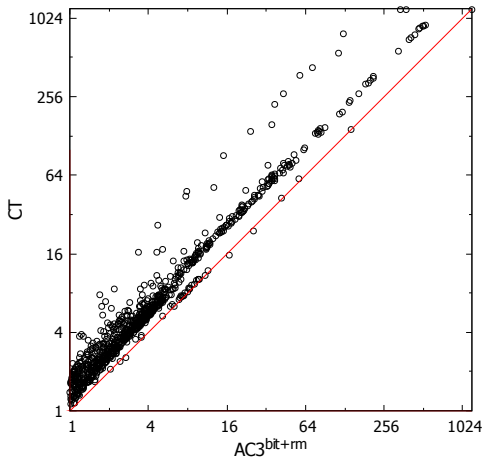
Table 5.1: Dual encoding memory size results: m/n means m instances were memory-out of n total, x% are fraction of original constraints.

can be much faster than STR2+ [Lec11] (see Chapter 3 for more details).

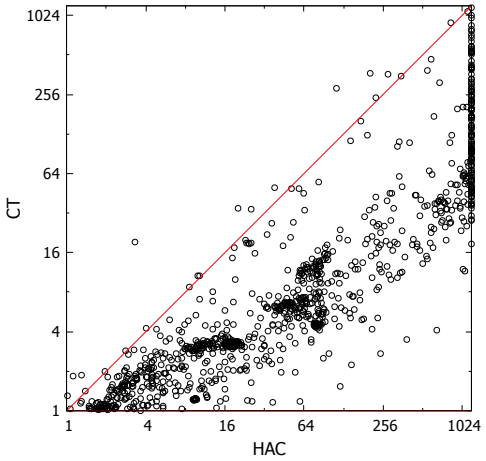
In order to understand and revisit the results from existing papers such as the ones above, we first test various kinds of CSP instances to compare different existing AC/GAC algorithms. Experimental details are given in Section 5.8 to avoid repetition. The main drawback of the dual encoding is the large size of binary constraints which lead to that the solver runs out of memory (more than 8 GB), which we call *memory-out*. We highlight that the memory is a more limiting resource than the CPU time. Details of the HVE and dual encodings are given in Chapter 4.

The relations which define the binary constraints from the dual encoding can be very large. This was also shown in [Lec11]. We also tested with various instances, Table 5.1, where the 100% column is the original CSP instance and the other p% column is the CSP instance generated by removing a random set of $(100 - p)$% constraints from the original CSP instance. The reduced CSP instances are essentially simpler than the original CSP instances, since a random set of constraints is removed. In Table 5.1, many CSP instances are simply memory-out, even if 80% constraints are removed, e.g., with Nonogram instances at 20% column (80% constraint removal) still 40 instances were memory-out. This confirms the result, i.e., the dual and double encoding are not practical, shown in [Lec11]. In this chapter, we focus on the HVE encoding as the dual encoding starts to become infeasible as the constraints become larger.

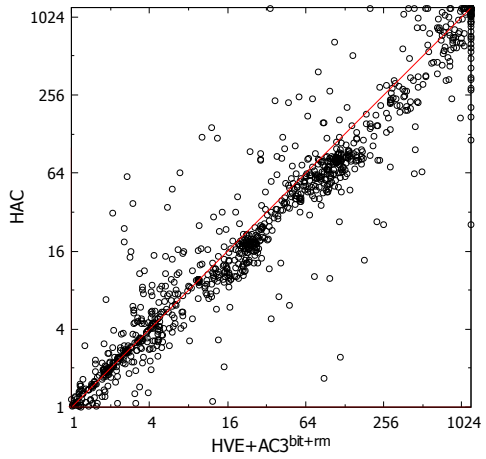
We revisit GAC vs AC with the HVE encoding given in Figure 5.1. Each dot in the graphs is a problem instance. For AC, we employ the state-of-the-art AC algorithm AC3^{bit}[LV08] with residues [LH07] denoted as AC3^{bit+rm}. It has been shown to be efficient in practice for AC because of the bit representation [LLY15b]. For GAC, we use CT which is one of



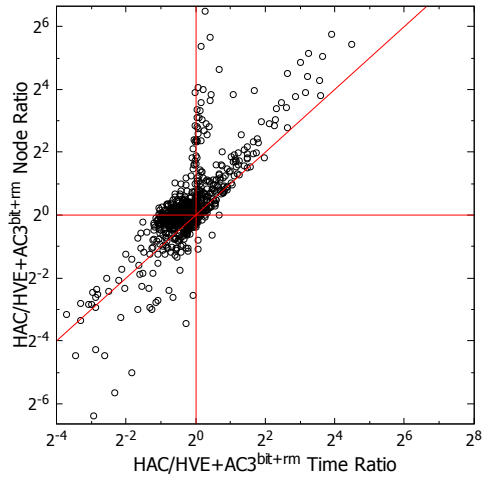
(a) AC3^{bit+rm} vs CT (binary CSPs)



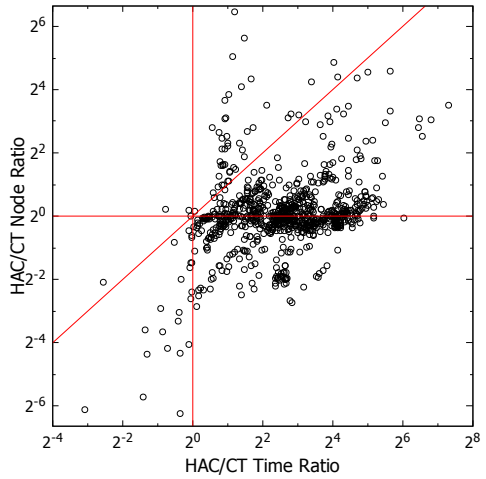
(b) HAC vs CT



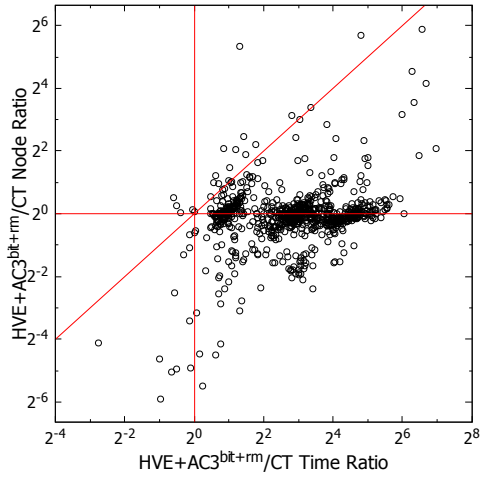
(c) HAC vs HVE+AC^{bit+rm}



(d) HAC vs HVE+AC^{bit+rm}(nodes+time)



(e) HAC vs CT(nodes+time)



(f) HVE+AC^{bit+rm} vs CT(nodes+time)

Figure 5.1: AC vs GAC algorithms: (a)-(c) use time on the axis while (d)-(f) use time and node ratios.

the state-of-the-art table GAC algorithms and was originally used in the Google OR-Tools solver.[2] Figure 5.1a compares the total runtime (in seconds) of AC^{bit+rm} with that of CT on binary CSPs. It shows that AC3^{bit+rm} can be faster than CT (a recent state-of-the-art table GAC algorithm) on binary CSPs. While it might not be surprising that a binary AC algorithm is faster than a non-binary GAC algorithm for binary constraints, it highlights the special nature of binary constraints. A special case of binary constraints is simpler than the non-binary case which typically has more complex algorithms and data structures. For example, the AC3^{bit+rm} algorithm in Figure 5.1a is much simpler than the CT algorithm and uses simpler data structures.

We now compare different ways of solving non-binary CSPs. We first compare the HAC algorithm on the HVE encoding with the CT algorithm on the original non-binary table constraints, where HAC is a specialized AC algorithm for the HVE encoding. Figure 5.1b compares the total runtime of HAC[3] with that of CT and shows that HAC is much slower than CT on non-binary CSPs. We highlight that this result is the opposite of the binary-only instances in Figure 5.1a and suggests what is known in the folklore that encoding a non-binary constraint to binary form to be solved using an AC propagator is much slower than directly using a (modern) GAC algorithm on the non-binary constraints. The HVE encoding with AC (using AC3^{bit+rm}) is also slow, shown in the comparison of time between HAC and HVE+AC in Figure 5.1c. Figure 5.1d compares time versus search nodes of HAC with HVE+AC with the y-axis giving the number of search nodes of HAC/(HVE+AC) and x-axis giving the total runtime of HAC/(HVE+AC), where the time (node) ratio of A/B means the ratio "the time (number of search nodes) of algorithm A" to "the total runtime (number of search nodes) of algorithm B". It shows that the specialized propagator of HAC for the HVE encoding is more efficient than AC3^{bit+rm} on the HVE encoding.

Figure 5.1e and 5.1f reveals important factors behind why the performance of the HVE encoding + AC algorithm is worse than GAC (with CT) on the non-binary instances:

(i) The concentration of points around the x-axis to the right of the y-axis shows that for a similar number of search nodes, the search time is significantly slower than CT (yet Figure 5.1a shows AC3^{bit+rm} is faster than CT for binary constraints which do not come from the hidden variable encoding). This suggests that the CT propagator is more efficient than the AC propagators on HVE instances;

(ii) There are many differences in the search nodes for the binary encoding versus the original instance. Many instances have more search nodes in the binary encoding as shown by the density of points above the x-axis.

[2] https://developers.google.com/optimization/cp/

[3] In this chapter, we use HAC to implicitly refer to HVE+HAC.

As many points are far to the right, we see that the propagator efficiency may be the factor for the superiority of CT though the difference in search nodes is also a factor.

Search heuristics and consistency propagators are the main components in a constraint solver. The results above show that with CT (and also other modern GAC propagators)[4] both the efficiency of the constraint propagator and effectiveness explain the basis for the folklore superiority of GAC on non-binary constraints over AC on binary encodings. Furthermore, we have seen that the binary constraints in the HVE encoding are very special. In this chapter, we focus on improving three problems identified for binary encoding instances:

1. Designing a more efficient AC propagator for the HVE encoding.

2. Avoiding making search heuristics on HVE worse than those on the original constraints.

5.3 Our method: a hidable model transformation propagator

As shown in Section 5.2, the results explain the folklore that it is better to handle non-binary constraints directly using GAC rather than using AC with binary encodings. The goal in this chapter is to dispel this folklore notion. We also see that binary encodings can interact poorly with the search heuristic and that the binary constraints in HVE are special.

To deal with the search heuristic problem, we "virtualize" the binary encoding so that the interaction between the binary encoded constraints can be hidden from the search heuristic making it behave like the search heuristic for the original non-binary constraint. This allows us to investigate the search heuristic, working on the HVE encoding, which (i) behaves like in the non-binary instance but also (ii) have an alternative. We incorporate ideas from modern GAC algorithms to get a more efficient AC propagator for the special kinds of binary constraints in HVE instances.

5.4 The hidable model transformation

The HVE encoding is a special transformation from a CSP P to another CSP P', which encodes each table constraint c_i as $|scp(c_i)|$ binary constraints s_i with a star structure. We generalize this idea to allow different kinds of transformations and search strategies.

Definition 5.1. *$(X_1 \cup X_2, C_2)$ is a* GAC hidable transformation *of (X_1, C_1) if C_2 has a partition $\{s_1, ..., s_m\}$ such that for each $c_i \in C_1$ and any c-tuple τ over $scp(c_i)$, $scp(c_i) \subseteq scp(s_i)$ and $rel(c_i) = sol(scp(s_i), s_i)[scp(c_i)]$ and enforcing GAC on $(scp(s_i), s_i)_{|\tau}$ can achieve GAC on the CSP $(scp(c_i), \{c_i\})_{|\tau}$, where $scp(s_i) = \bigcup_{c \in s_i} scp(c)$.*

[4] Experimental results for STR2+ are also similar to CT and have not been shown.

In the definition above, the c-tuple τ is used to denote current variable domains, and the notation $(X, C)_{|\tau}$ denotes a sub-problem of a CSP (X, C) by setting variable domains to τ (introduced in Section 2.1). In addition, each set s_i of constraints can be regarded as a special constraint modelling the original constraint c_i such that enforcing GAC on the constraints in s_i with current variable domains can achieve GAC on c_i. Therefore, we can directly get the following result.

Corollary 5.1. *If $P' = (X_1 \cup X_2, C_2)$ is a GAC hidable transformation of a CSP $P = (X_1, C_1)$, enforcing GAC on $P'_{|\tau}$ can achieve GAC on $P_{|\tau}$, where τ is a c-tuple over X_1.*

If $P' = (X_1 \cup X_2, C_2)$ is a GAC hidable transformation of P, then the solver only needs to do search on X_1, since enforcing GAC on P_2 can check whether an assignment on X_1 is a solution of P, where the GAC propagators on P' are a "black box".

Corollary 5.2. *The HVE encoding $(X \cup H, HC)$ is a GAC hidable model transformation.*

Proof. For any constraint c_i, we can set $s_i = \{c_i^x \in HC | x \in scp(c_i)\}$, since $(scp(s_i), s_i)$ is the HVE of $(scp(c_i), \{c_i\})$ where s_i has a star structure. The HVE encoding by construction of c_i^x already meets the requirements of Definition 5.1. □

The HVE encoding is a GAC hidable model transformation, therefore, enforcing AC on the HVE encoding can achieve GAC on the original constraints, and search algorithms only need to consider the original variables. In addition, if the HVE encoding is not used to transform a constraint c_i, then s_i can be set to $\{c_i\}$, e.g., it does not need to employ the HVE encoding on binary constraints.

5.5 A propagator for the hidable model transformation

A hidable binary encoding transforms constraints into binary constraints so in principle any AC propagator can be used, however, we propose a more refined strategy. Algorithm 5.1 gives the *HTAC* algorithm to enforce AC/GAC on GAC hidable binary encoding instances. HTAC adds a variable $x \in X_1 \cup X_2$ to a propagation queue $\mathcal{Q}$ if x may be used to update the domains of other variables. Then HTAC iteratively picks a variable x from $\mathcal{Q}$, and then use AC/GAC algorithms to enforces AC/GAC on all constraints in every subset $s_i \in \mathcal{S}$ such that $x \in scp(s_i)$. For different s_i, we can use different AC/GAC algorithms. If c_i is a constraint in C_1 and $s_i = \{c_i\}$, i.e., c_i is not transformed into other constraints, then HTAC uses AC/GAC algorithms to directly enforce GAC on c_i. This allows us to only employ the HVE encoding on non-binary table constraints. In addition, if $|s_i| > 1$, then we can use specialized AC-H algorithms, exploiting the nature of binary encodings, to enforce AC on all binary constraints in the subset s_i. In section 5.6, we present a specialized AC-H algorithm for the HVE transformation.

Algorithm 5.1: HTAC (X_1, C_1)

```
1  let (X_1 ∪ X_2, C_2) be the hidable model transformation of (X_1, C_1)
2  let S be a partition of C_2 making (X_1 ∪ X_2, C_2) hidable
3  Q ← X_1
4  while Q ≠ ∅ do
5      pick and delete x from Q
6      for s_i ∈ S s.t. x ∈ scp(s_i) do
7          if |s_i| = 1 then
               // s_i = {c_i}
8              if ¬GAC(s_i) then
9                  return false
10             end
11         end
12         else if ¬AC-H(s_i) then
13             return false
14         end
15     end
16 end
17 return true
```

The HTAC algorithm is different from HAC [SS05]: (i) HTAC adds original variables to the propagation queue $\mathcal{Q}$ while HAC only adds hidden variables to $\mathcal{Q}$. When the domain of a variable x is changed, HAC directly updates the domains of all hidden variables constrained by x and does not add x to $\mathcal{Q}$; (ii) HTAC uses a reversible bit set to represent the domain of a hidden variable (see Section 5.6) but HAC uses an ordered linked set; (iii) The revise functions used in AC-H are also different from HAC.

For a GAC hidable model transformation $(X_1 \cup X_2, C_2)$, the solver only needs to search the variables in X_1. Variable search heuristics are an important part of the solver, in this section, we will simply say "heuristic" for the variable search heuristics. The search heuristics using information from the constraint structure can choose to use the structure of the GAC hidable model transformation or the original model. For example, the wdeg/dom [BHLS04] heuristic[5] records a weight w for each constraint, and increasing w by 1 if the constraint

[5] wdeg/dom is among the best general purpose variable search heuristics [WLPT19].

causes inconsistency. Variables are selected based on the constraint weights. For the HVE transformation, we propose two alternatives for wdeg/dom:

A. using wdeg/dom with the original model, we record a weight w_i for each $s_i \in \mathcal{S}$. Thus, w_i is regarded as a weight for a virtual constraint representing the weight of $c_i \in C_1$. Weight w_i is incremented if AC(-H)(s_i) is not consistent;

B. using wdeg/dom with HVE, we record a weight w_i^x for each $c_i^x \in C_2$ and w_i^x is incremented if AC(-H)(s_i) is not consistent, where x is picked from $\mathcal{Q}$ and $x \in scp(s_i)$.

We call HTAC as *HTAC1* if the heuristics use the original non-binary model (A); and *HTAC2* if the heuristics use the transformation model (B).

5.6 The AC-H algorithm for HVE

We first introduce the data structures used in the algorithm which incorporates data structures used by (modern) GAC and AC algorithms [LS06, Lec11, LV08, DHL+16]:

1. For a original variable x:
   - $dom(x)$ uses an ordered linked set (see Chapter 2.5) to represent the domain of x.
   - $bitDom(x)$ uses a bit set (see Chapter 2.5) to represent the domain of x.
   - $bitSup(c, x, a)$ is used to record all supports in $bitDom(y)$ of a variable value $a \in \mathcal{D}(x)$, where $scp(c) = \{x, y\}$ [LV08], and $rm[c, x, a]$ records a word in $bitDom(y)$ which may include supports of (x, a).
   - $lastSize(c, x)$ is used to record the size of $dom(x)$ after the last update on the domain of y based on c [Lec11], and $dn = lastSize(c, x) - dom(x)$ is the number of values deleted since the last time updating y on c, where $scp(c) = \{x, y\}$.
2. For a hidden variable x:
   - $bitDom(x)$ uses a bit set to represent the domain of x, if $bitDom(x)$ is changed, the old states of $bitDom$ are recorded in a stack so that it can be undone on backtracking. In the algorithm, the state saving is implicit and not part of the algorithm to simplify the description.
   - $wordDom(x)$ is a sparse set (see Chapter 2.5) used to record the non-ZERO words in $bitDom(x)$. It uses the reversible sparse bit-set idea in [DHL+16].
   - $prevDom(x)$ is a copy of $bitDom(x)$, we use $prevDom(x)$ to check whether $bitDom(x)$ is changed.
   - $buf0$ is a bit set, where all words in $buf0$ are initialized as ZERO (ZERO is the bit set with all zeroes).

Algorithm 5.2: AC-H (s_i)

```
1  let hv_i be the hidden variable constrained with binary constraints in s_i
2  for each c_i^x ∈ s_i do
3  |  revise2(c_i^x, hv_i)
4  end
5  if ¬update(hv_i) then
6  |  return false
7  end
8  for each c_i^x ∈ s_i do
9  |  if revise1(c_i^x, x) then
10 |  |  Q ← Q ∪ {x}
11 |  end
12 end
13 return true
```

Algorithm 5.2 is to enforce AC on a set s_i of constraints for HVE instances, where $s_i = \{c_i^x \in HC | hv_i \in scp(c_i^x)\}$. The HVE transformation has a star structure constraint graph (a special tree). This allows the AC-H algorithm to update the domains of variables in two passes: (i) from leaves ($x \in c_i$) to the root (hv_i), and (ii) from the root to the leaves.

The first phase of revise is with function *revise2* to (partially) update the domain of hv_i based on the domains of variables in $scp(c_i)$. Then function *update* updates $wordDom$ representation of hv_i. If $wordDom(hv_i) = \emptyset$, the instance is not AC. The second revise phase uses function *revise1* to update the domains of all variables in $scp(c_i)$. If the domain of a variable x is changed, x is added to Q. We do not add hv_i to Q, since the domains of variables constrained with hv_i are updated in the AC-H algorithm.

The overall structure of the AC-H algorithm is similar to any AC algorithm using revise functions except that we exploit the star constraint graph as explained above. The domain of hv_i is only updated by the function *revise2*. Meanwhile, the function *revise2* deletes the values which do not have valid supports in the domains of the variables in $scp(c_i)$ from the domain of hv_i. If the function *update* returns false, then all words in $bitDom(hv_i)$ become ZERO, which means that it is not AC on the binary constraints in s_i. In addition, the function *revise1* is similar to AC3^{bit+rm}.

Function $revise1(c, x)$

```
1  size ← |dom(x)|
2  for each a ∈ dom(x) do
3  |  if ¬seekSupport(c, x, a) then remove a from dom(x)
4  end
5  return |dom(x)| ≠ size
```

Function $revise2(c, hv)$

```
1  Let x be the variable such that scp(c) = {x, hv}
2  dn ← lastSize(c, x) − |dom(x)|
3  if dn > |dom(x)| then reset(c, hv)
4  else if dn > 0 then
5  |  delete(c, hv, dn)
6  end
```

Function $seekSupport(c, x, a)$

```
1  Let y be the variable such that scp(c) = {x, y}
2  w ← rm[c, x, a]
3  if (bitSup[c, x, a][w] & bitDom[y, w]) ≠ ZERO then return true
4  for each w ∈ wordDom(y) do
5  |  if (bitSup[c, x, a][w] & bitDom[y, w]) ≠ ZERO then
6  |  |  rm[c, x, a] ← w
7  |  |  return true
8  |  end
9  end
10 return false
```

Function $delete(c, hv, dn)$

```
1  Let x be the variable such that scp(c) = {x, hv}
2  for i = 0 to dn do
3  |  Let a be the last i value deleted from dom(x)
4  |  for each w ∈ wordDom(hv) do
5  |  |  bitDom[hv, w] ← bitDom[hv, w] & ¬bitSup[c, x, a]
6  |  end
7  end
```

Function $reset(c, hv)$

```
1  Let x be the variable such that scp(c) = {x, hv}
2  for each a ∈ dom(x) do
3  |  for each w ∈ wordDom(hv) do
4  |  |  buf0[w] ← buf0[w] | bitSup[c, x, a]
5  |  end
6  end
7  for each w ∈ wordDom(hv) do
8  |  bitDom[x, w] ← buf0[w] & bitDom[hv, w]
9  |  buf0[w] ← ZERO
10 end
```

Function $update(x)$

```
1  for each w ∈ wordDom(x) do
2  |  if bitDom[x, w] ≠ prevDom[x, w] then
3  |  |  save prevDom[x, w] in a stack
4  |  |  prevDom[x, w] ← bitDom[x, w]
5  |  |  if bitDom[x, w] = ZERO then
6  |  |  |  remove w from wordDom(x)
7  |  |  end
8  |  end
9  end
10 return wordDom(x) ≠ ∅
```

5.7 Revise functions

In AC algorithms, the $revise(c, x)$ functions are used to update $dom(x)$ based on $dom(y)$, where $\{x, y\} = scp(c)$, i.e removing the values in $dom(x)$ which don't have valid supports in $dom(y)$. We give two revise functions used in AC-H to handle the reversible bit-set domains. The function *revise1* updates original variable domains using function *seekSupport* which is similar to that in AC3^{bit+rm} algorithm. The difference is that our *seekSupport* only check the words in *wordDom*. For hidden variables, we do not use the *seekSupport* function, since each value in hidden variable domains (by construction) only has one support in $rel(c)$, i.e., an hidden variable hv functionally determines the values in the domains of original variables constrained with hv (see [BVB98]), which make *bitSup* useless. Using the ideas from GAC algorithms CT[DHL+16] and STRbit[WXYL16], function *revise2* applies the *delete* and *reset* functions to update *bitDom*. If the number of values deleted from $dom(x)$ is larger than $|dom(x)|$, then function *delete* is used to remove all supports of the deleted values from

$bitDom(hv)$. Otherwise, function *reset* is used to build a new $bitDom(hv)$ based on current $dom(x)$. After updating $bitDom(hv)$ of a hidden variable hv, the function *update* is used to check $bitDom(hv)$ for domain wipeout, for each word w in $bitDom(hv)$, if w is changed, the old value of $preDom(w)$ in saved on a stack for backtracking, and if $w = ZERO$ it is removed from $wordDom$.

5.8 Experiments

We present experiments to evaluate HTAC on the hidden variable encoding (HVE+HTAC1 and HVE+HTAC2) compared with HVE+AC3^{bit+rm}, HAC, STR2+, CT and STRbit algorithms. HTAC, HVE+AC3^{bit+rm} and HAC maintain AC (MAC) on the hidden variable encoding instances. STR2+, CT and STRbit maintain GAC (MGAC) on the original non-binary instances. CT and STRbit are the state-of-the-art GAC algorithms, and HAC is the best existing algorithm for binary encoding from Section 5.2. We used binary and non-binary instances from the XCSP3 website (http://xcsp.org). Instances from XCSP3 which timeout for all compared algorithms are ignored. In total, we evaluated 1328 binary and 2431 non-binary problem instances. The binary CSP series are:

> QCP, QWH, Geometric, Rlfap, Driver, Lard, Queens, RoomMate, PropStress, QueensKnights, KnightTour, Random

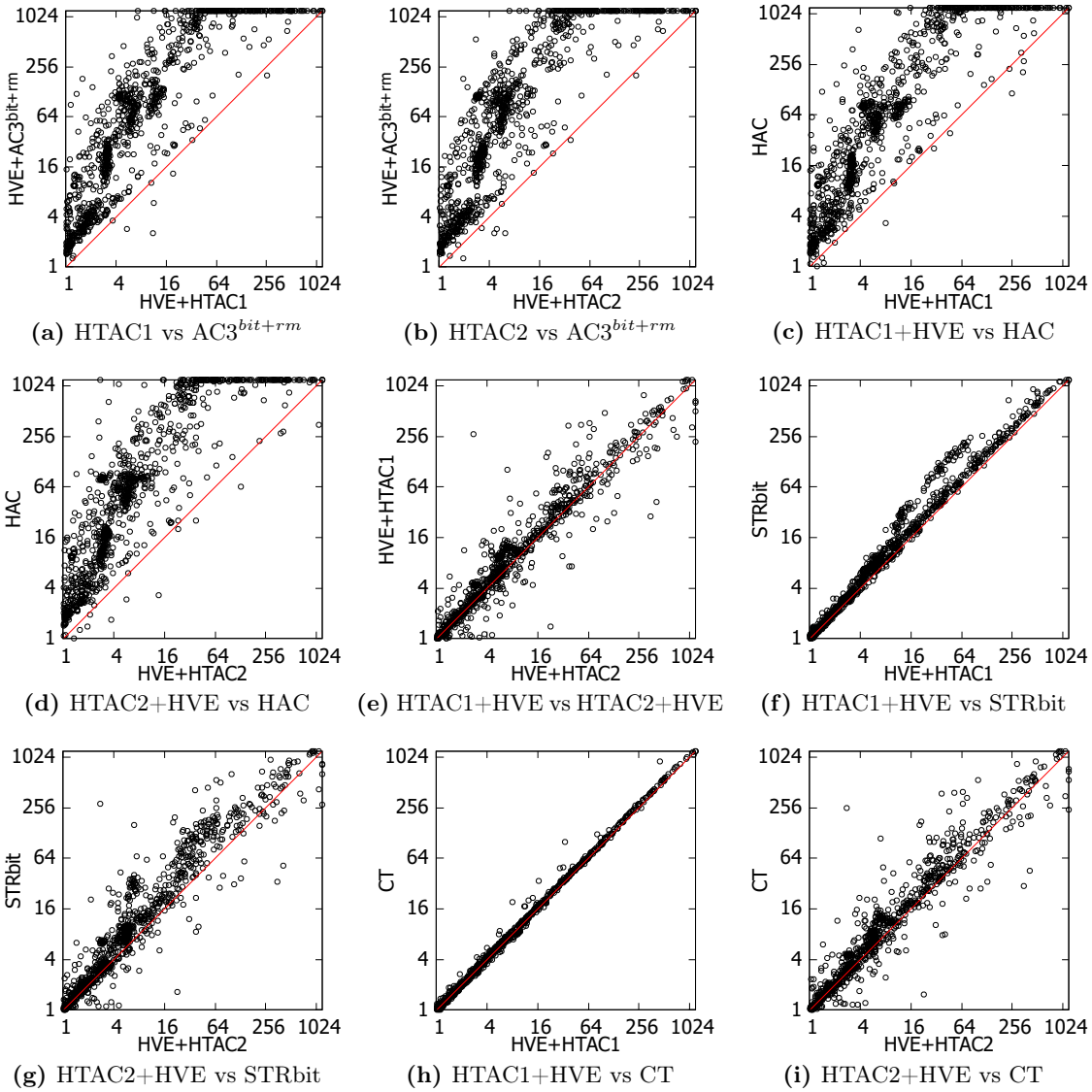
Results on the binary instances are discussed in Section 5.2. The non-binary CSP series are:

> Kakuro, Dubois, PigeonsPlus, MaxCSP,Renault, Aim, Jnh, Cril, Tsp, Various, Nonogram, Bdd-{15,18}-21, mdd-7-25-{5,5-p7,5-p7}, reg2ext,Rand-3-24-24, Rand-3-24-24f, Rand-5-12-12, Rand-5-{2,4,8}X, Rand-10-20-10, Rand-10-20-60, Rand-15-23-3, Rand-5-10-10, Rand-5-12-12t, Rand-7-40-8t, Rand-8-20-5, Rand-3-20-20, Rand-3-20-20f

This section focuses on the non-binary instances. Results on the binary series are also given in Section 5.2 comparing HAC, HVE+AC3^{bit+rm} and CT. The experiments were run on a 3.20GHz Intel i7-8700 machine. We implemented HTAC in the Abscon solver[6] which has the other algorithms implemented. In addition, we optimized the Abscon CT and HAC implementation to be a little faster.[7] The variable search heuristic used is $wdeg/dom$ and the value heuristic used is lexical value order. The $wdeg/dom$ variable heuristic with restart is considered one of state-of-the-art heuristics in classic constraint solvers [HS17]. The restart

[6] https://www.cril.univ-artois.fr/%7Elecoutre/#/softwares

[7] While implementing HTAC, we found some optimizations for the existing CT and HAC code. This gives some advantages to CT and HAC for Abscon.



(a) HTAC1 vs AC3^{bit+rm} **(b)** HTAC2 vs AC3^{bit+rm} **(c)** HTAC1+HVE vs HAC

(d) HTAC2+HVE vs HAC **(e)** HTAC1+HVE vs HTAC2+HVE **(f)** HTAC1+HVE vs STRbit

(g) HTAC2+HVE vs STRbit **(h)** HTAC1+HVE vs CT **(i)** HTAC2+HVE vs CT

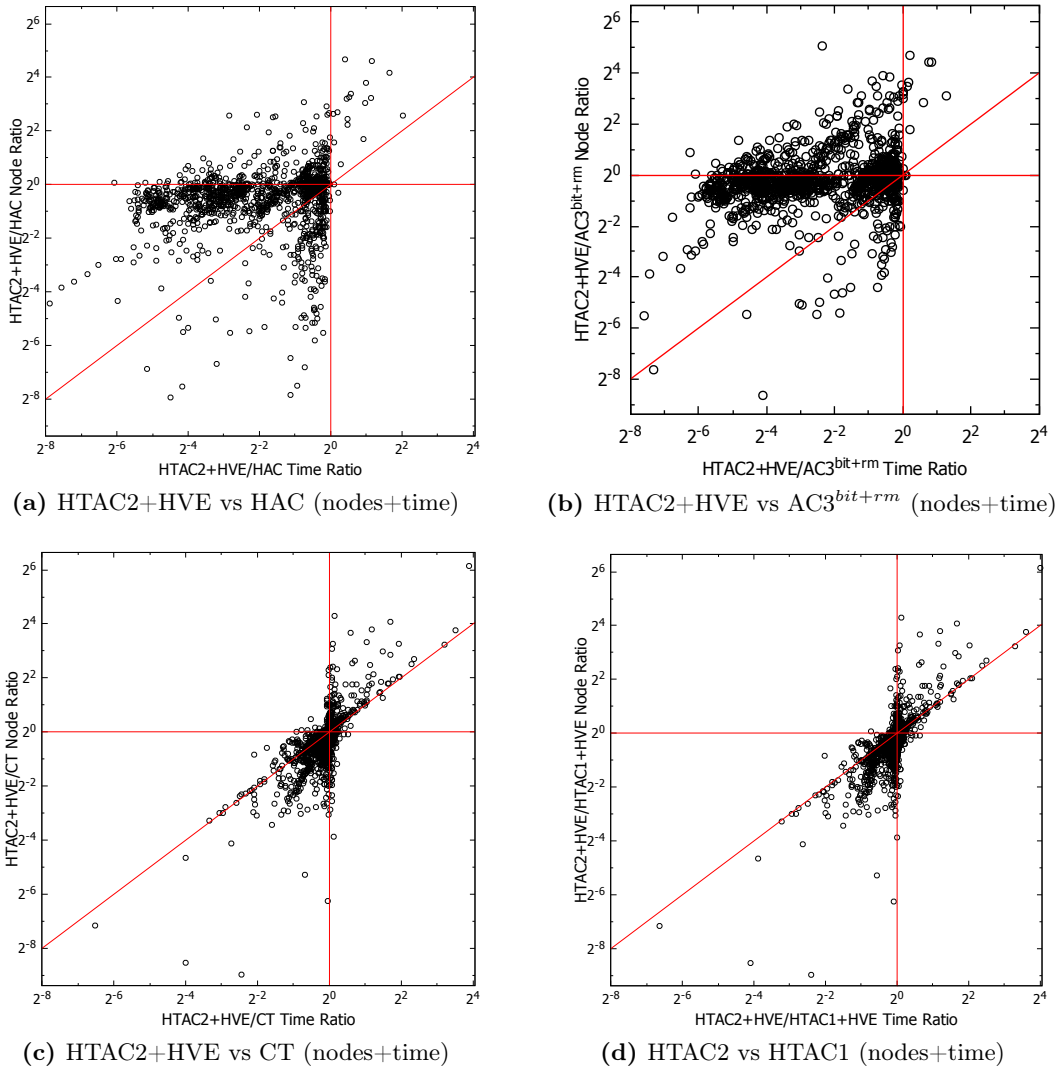
Figure 5.2: HTAC vs other algorithms: using time on the axis.

policy was geometric restart (the initial *cutoff* = 10 and $\rho = 1.1$)[8]. CPU time is limited to 1200 seconds per instance and memory to 8GB.

Figure 5.2 (time comparison) and Figure 5.3 (time + node comparison) show scatter plots to compare HVE+HTAC with other algorithms. Each dot in the plots is a CSP instance. Figures 5.2a to 5.2i compare the runtime[9] of different algorithms. A dot above (below) the diagonal means the algorithm on x-axis (y-axis) is faster on the corresponding

[8] *cutoff* is the allowed number of failed assignments for each restart. After restart, *cutoff* increases by (*cutoff* $\times$ ρ)

[9] For binary encoding instances, the total runtime includes model transformation time and solving time.



(a) HTAC2+HVE vs HAC (nodes+time)

(b) HTAC2+HVE vs AC3^{bit+rm} (nodes+time)

(c) HTAC2+HVE vs CT (nodes+time)

(d) HTAC2 vs HTAC1 (nodes+time)

Figure 5.3: HTAC vs other algorithms: using node and time ratios on the axis.

instance. Meanwhile, Figures 5.3a, 5.3c, 5.3d and 5.3b compare the time ratio and node ratio, where the time (node) ratio of A/B means the ratio "the total runtime (number of search nodes) of algorithm A" to "the total runtime (number of search nodes) of algorithm B". Figures 5.2a,5.2b (HTAC vs AC3^{bit+rm}) and Figures 5.2c, 5.2d (HTAC vs HAC) show that HTAC can substantially outperform existing AC algorithms on HVE instances. From Figure 5.3a, giving the time ratio and node ratio of (HTAC2+HVE)/HAC (see the discussion of ratio graphs in Section 5.2), we see that HTAC is generally much faster than HAC, since most points around the x-axis are at the left of the y-axis. For most instances, the search nodes of HTAC2 is less than HAC. Figure 5.3b shows the time ratio and node ratio of (HTAC2+HVE)/AC3^{bit+rm}.

Figures 5.2f, 5.2g (HTAC vs STRbit) and Figures 5.2h and 5.2i (HTAC vs CT) show HTAC

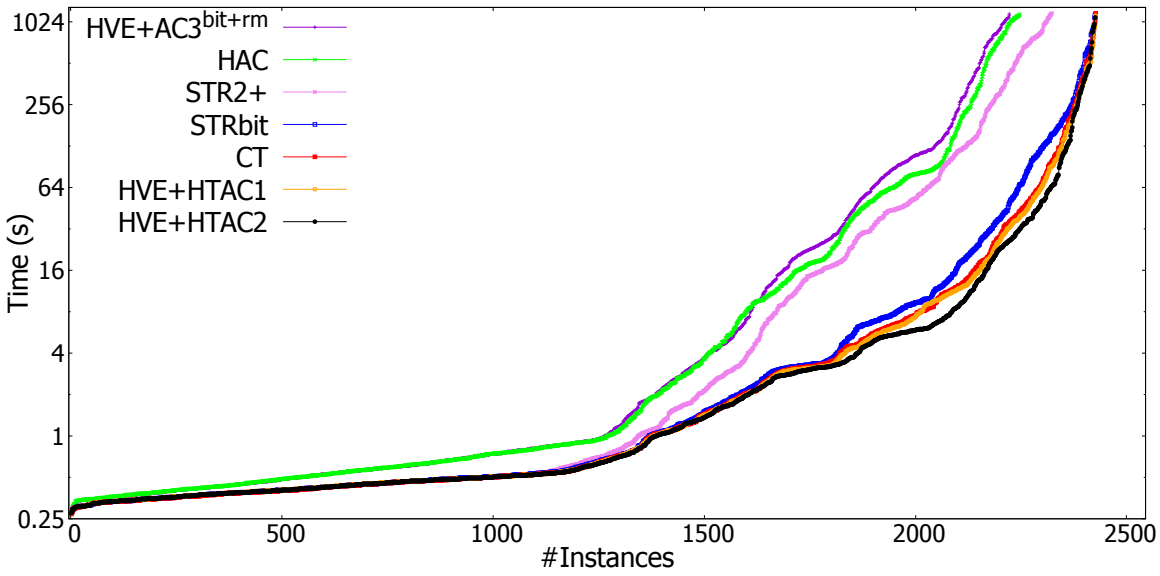


Figure 5.4: Runtime Distribution: HTAC, HAC, HVE+AC3^{bit+rm}, CT, STRbit, STR2+.

is competitive with the state-of-the-art GAC algorithms CT and STRbit. HTAC1+HVE using wdeg/dom on the original model is faster than STRbit and competitive with CT, being faster than CT on some instances. HTAC2+HVE using wdeg/dom on the HVE transformed model is faster than STRbit and CT on many instances. Figure 5.3c combines node ratio and time ratio to show the runtime and search nodes trade-offs of HTAC2 with CT with more instances having less nodes and time. Figures 5.2e, 5.3d compare HTAC2 with HTAC1, the performance of HTAC1 is similar to CT.

Figure 5.4 (in color) shows the runtime distribution of the problem instances solved by the different algorithms. The y-axis is the total runtime (in seconds) and the x-axis is the number of instances solved within the time limit. The algorithm corresponding to the line at the bottom right corner has the best performance. Figure 5.4 explains why the (incorrect) folklore has developed, binary encodings are slower with HVE+AC3^{bit+rm} and HAC being behind STR2+ (also show in [Lec11]). There is a separation between STR2+ and newer GAC algorithms (STRbit and CT). The performance of our HTAC algorithms is competitive with STRbit and CT, in particular, HTAC2 is faster on some instances.

5.9 Conclusion

We first show experimental results which can explain the folklore that it is better to handle a non-binary constraints directly with GAC than by a binary encoding of the non-binary constraints and using AC. We believe this is the first work which shows that this folklore is incorrect and misleading. Encoding a non-binary CSP with the existing HVE binary

encoding but with an improved propagator and the hidable transformation to handle the search strategy can be competitive with using GAC on the original non-binary CSP.

We propose a new propagator HTAC. By using the GAC hidable binary encoding with HTAC, we can address the differences in search nodes so that the search space on the binary instance behaves as in the non-binary instance but it also allows search on the binary encoded model. The HTAC propagator gains efficiency by using properties of binary constraints in the HVE. It is also efficient as we apply ideas from modern GAC algorithms. Experiments show that HTAC on the binary encoded instance is competitive with state-of-the-art table GAC algorithms on the original instances, in some cases, HTAC is faster. Not only have we shown that solving with the binary encoding is viable and competitive, we believe that it opens new directions for modelling and solver algorithms while still retaining the original non-binary instance. Binary instances and constraints are special being simpler so the algorithms can also be simpler, which also aids in efficiency.

Chapter 6

Bipartite encoding

6.1 Introduction

Bit representation is the key to the efficiency of many state-of-the-art Generalized Arc Consistency (GAC) algorithms for table constraints. A literature review on table GAC algorithms can be found in Chapter 3. Recently, bit representation has been extended to handle compact representations and high-order consistencies. However, experiments show that the CT algorithm is still overall faster than these newer algorithms (see the experimental results of PW-CT [SC18], compact-MDD [VLS18] and smart MDD [VLS19]). We will show that the CT algorithm can be improved from a binary encoding perspective.

The Hidden Variable Encoding (HVE) [RPD90] and dual encoding [DP89] are 2 well known binary encodings for transforming table constraints into binary constraints. Arc Consistency (AC) on the dual encoding can be stronger than AC on the HVE encoding [BSW08]. However, the dual encoding is much larger than the HVE encoding. The HVE encoding, proposed 30 years ago, is still the state-of-the-art binary encoding of table constraints (see Chapter 5). We observe that the binary constraints constructed in binary encodings have special structures. Specialized AC algorithms, such as HAC [MS01], PW-AC [SS05] and HTAC (Algorithm 5.1), have been proposed to enforce AC on binary encodings. In particular, HTAC is competitive with CT, suggesting that AC algorithms on binary encodings have the potential to outperform the state-of-the-art table GAC algorithms.

In this chapter, we propose a new binary encoding, called Bipartite Encoding (BE), to encode constraints as binary constraints between factor variables partitioning constraint scopes. Unlike the HVE encoding, AC on the BE encoding is stronger than GAC on the original constraints. We then give an algorithm to construct the BE encoding of table constraints, followed by an AC algorithm AC-BE which treats each connected component in the BE encoding as a "special constraint". We evaluate AC-BE to handle table constraints comparing with state-of-the-art table GAC algorithms and binary encodings, namely, CT, STRbit and HVE (with HTAC). Our experiments show that BE is the first binary encoding which can significantly outperform the CT, STRbit and HTAC algorithms. The usefulness of stronger consistency is also shown to be effective in making some instances backtrack free.

6.2 Bipartite Encoding

We start with an observation that every binary constraint is over 2 variables which partition the constraint scope. So we can think of a binary constraint as being encoded with such a partitioning. We generalize this idea to design a new binary encoding called Bipartite Encoding (BE) which can further give a consistency that is stronger than GAC.

Definition 6.1. *Given a CSP $P = (X, C)$, a* factor variable *fv over a non-empty set of variables $S \subseteq X$ is a variable such that (i) if $|S| > 1$, $\mathcal{D}(fv)$ is a set of tuples over S and $\tau[S] \in \mathcal{D}(fv)$ for all solutions τ of P, otherwise (ii) fv is the original variable in S. We use $scp(fv) = S$ to denote the scope of the variables covered by fv. Then a factor variable fv is* original *if $|scp(fv)| = 1$ and* compound *if $|scp(fv)| > 1$.*

The minimum domain of a compound factor variable fv consists of all tuples τ over $scp(fv)$ which can be extended to solutions of the CSP. It is a empty set if and only if the CSP is unsatisfiable. As such, it is NP-hard to find the minimum domain of fv. We propose to use local consistent tuples to construct $\mathcal{D}(fv)$ (see Section 6.3).

Definition 6.2 (Bipartite Encoding)**.** *A* bipartite encoding *$BE(P)$ of a CSP $P = (X, C)$ is a CSP $(X \cup X^*, C^+ \cup C^*)$, where*

- *X^* and X respectively denote the compound and original factor variables, and for all variables $fv_i, fv_j \in X^*$, if $fv_i \neq fv_j$, then $scp(fv_i) \neq scp(fv_j)$.*
- Partition constraints *$C^* = \{c^* | c \in C\}$: each $c^* \in C^*$ is a binary constraint such that*
  - *$scp(c^*) = \{v_1, v_2\}$ and $scp(c^*) \subseteq X \cup X^*$,*
  - *$\{scp(v_1), scp(v_2)\}$ is a partition of $scp(c)$,*
  - *$rel(c^*) = \{\{(v_i, a_i), (v_j, a_j)\} | a_i \in \mathcal{D}(v_i), a_j \in \mathcal{D}(v_j), (t(v_i, a_i) \cup t(v_j, a_j)) \in rel(c)\}$ and*
  - *$t(v, a) = a$ if $v \in X^*$, otherwise $t(v, a) = \{(v, a)\}$.*
- Mapping constraints *$C^+ = \{c_i^x | fv_i \in X^*, x \in scp(fv_i)\}$: for each $c_i^x \in C^+$, $scp(c_i^x) = \{fv_i, x\}$ and $rel(c_i^x) = \{\{(fv_i, \tau), (x, a)\} | \tau \in \mathcal{D}(fv_i), a \in \mathcal{D}(x), (x, a) \in \tau\}$.*

For all compound factor variables $fv_i \in X^*$, the information of the tuples (values) in the domain $\mathcal{D}(fv_i)$ is recorded in the mapping constraints $\{c_i^x | x \in scp(fv_i)\}$, which ensures that $t(v_1, a_1) \cup ... \cup t(v_m, a_m)$ is an assignment over original factor variables if the tuple $\{(v_1, a_1), ...(v_m, a_m)\}$ is consistent on $BE(P)$, where $t(v, a) = a$ if v is a compound factor variable, otherwise $t(v, a) = \{(v, a)\}$. We remark that the mapping constraints are similar to the binary constraints used in the HVE encoding. Additionally, for each partition constraint $c^* \in C^*$, a tuple $\{(v_i, a_i), (v_j, a_j)\}$ in $rel(c^*)$ corresponds to the tuple $t(v_i, a_i) \cup t(v_j, a_j)$ in $rel(c)$, which ensures that the tuple $t(v_1, a_1) \cup ... \cup t(v_m, a_m)$ is a solution of P if the tuple $\{(v_1, a_1), ...(v_m, a_m)\}$ is a solution of $BE(P)$.

x_1	x_2	x_3	x_4
0	0	0	1
0	0	1	0
0	1	0	0
1	0	0	1

(a) c_1

x_1	x_2	x_5	x_6
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	1	0	0

(b) c_2

fv_1	fv_2
0	1
0	2
1	0

(c) c_1^*

fv_1	fv_3
0	1
0	3
1	1
1	2

(d) c_2^*

fv_1	x_1
0	0
1	0

(e) $c_1^{x_1}$

fv_1	x_2
0	0
1	1

(f) $c_1^{x_2}$

fv_2	x_3
0	0
1	0
2	1

(g) $c_2^{x_3}$

fv_2	x_4
0	0
1	1
2	0

(h) $c_2^{x_4}$

fv_3	x_5
1	0
2	1
3	1

(i) $c_3^{x_5}$

fv_3	x_6
1	1
2	0
3	1

(j) $c_3^{x_6}$

Figure 6.1: A bipartite encoding example.

Example 6.1. *Let P be a CSP (X, C) where C=$\{c_1, c_2\}$, X=$\{x_1, \cdots, x_6\}$ and domains are $\{0, 1\}$ and $scp(c_1)$=$\{x_1, \cdots, x_4\}$ and $scp(c_2)$=$\{x_1, x_2, x_5, x_6\}$ with relations given in Figure 6.1a, 6.1b. A BE $(\{fv_1, fv_2, fv_3, x_1, \cdots, x_6\}, \{c_1^*, c_2^*, c_1^{x_1}, \cdots, c_3^{x_6}\})$ of P is shown in Figures 6.1c,$\cdots$,6.1j, where Figures 6.1c,6.1d are the partition constraints c_1^*, c_2^* and Figures 6.1e, $\cdots$, 6.1j are the mapping constraints $c_1^{x_1}, \cdots, c_3^{x_6}$, respectively. c_1 and c_2 are represented as c_1^* and c_2^*, where $scp(c_1^*) = \{fv_1, fv_2\}$ and $scp(c_2^*) = \{fv_1, fv_3\}$. fv_1, fv_2 and fv_3 are factor variables on $S_1 = \{x_1, x_2\}$, $S_2 = \{x_3, x_4\}$ and $S_3 = \{x_5, x_6\}$. For each factor variable on $\{x, y\}$, we use values 0, 1, 2 and 3 to denote tuples $\{(x, 0), (y, 0)\}$, $\{(x, 0), (y, 1)\}$, $\{(x, 1), (y, 0)\}$ and $\{(x, 1), (y, 1)\}$, respectively. Each solution of $BE(P)$ corresponds to a solution of P, e.g., $\{(fv_1, 0), (fv_2, 1), (fv_3, 3), (x_1, 0), (x_2, 0), (x_3, 0), (x_4, 1), (x_5, 1), (x_6, 1)\}$ corresponds to $\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 1), (x_5, 1), (x_6, 1)\}$.*

Proposition 6.1. *A CSP $P = (X, C)$ is GAC if $BE(P) = (X^* \cup X, C^* \cup C^+)$ is AC.*

Proof. Given any $c \in C$ and $v_i \in scp(c^*)$ and $x \in scp(v_i)$ and $a \in \mathcal{D}(x)$ where $scp(c^*) = \{v_i, v_j\}$. $BE(P)$ is AC, hence, there is $a_i \in \mathcal{D}(v_i)$ such that $(x, a) \in t(v_i, a_i)$ and a_i has a support $a_j \in \mathcal{D}(v_j)$. Then $t(v_i, a_i) \cup t(v_j, a_j)$ is a valid support of (x, a) on c. So all constraints $c \in C$ are GAC and P is GAC. □

AC on $BE(P)$ can be stronger than GAC on P (see Proposition 6.2 in Chapter 6.3.1). For example, $(x_1, 1)$ is not AC on the BE given in Figure 6.1 but the original CSP is GAC. In addition, we remark that the bipartite encoding can be used to handle a subset C_1 of the constraints in a CSP $P = (X, C_1 \cup C_2)$ by encoding P as a CSP $(X \cup X^*, C_1^* \cup C_1^+ \cup C_2)$ such that $(X \cup X^*, C_1^* \cup C_1^+)$ is a BE of the CSP (X, C_1), in case it is too expensive to encode all constraints as binary constraints.

Algorithm 6.1: BE(X,C)

1 $scpp \leftarrow partition(C)$
2 $S \leftarrow \bigcup_{c \in C} scpp[c]$, $V \leftarrow \emptyset$
3 **for** $S_j \in S$ **s.t.** $|S_j| > 1$ **do**
4 Generate a factor variable fv_j & domain on S_j
5 $V \leftarrow V \cup \{fv_j\}$
6 **end**
7 $V \leftarrow V \cup X$, $C^* \leftarrow \emptyset$
8 **for** $c \in C$ **do**
9 $\{S_i, S_j\} \leftarrow scpp[c]$
10 Construct c^* between v_i and v_j such that $scp(v_i) = S_i$ and $scp(v_j) = S_j$
11 $C^* \leftarrow C^* \cup \{c^*\}$
12 **end**
13 Construct constraints C^+ with the factor variables in V
14 **return** $(V, C^* \cup C^+)$

6.3 A Bipartite Encoding algorithm

We present Algorithm 6.1 for constructing a bipartite encoding instance from a CSP (X, C). It is based on partitions of constraint scopes and has three parts given in Algorithm 6.1:

(i) For each constraint $c \in C$, the partition function split the constraint scope $scp(c)$ into 2 subsets $scpp[c]$ (Line 1—see Section 6.3.1).

(ii) Generating compound factor variables (at Lines 2-6). We set the domain of a compound factor variable fv as the projection $\bigcap\{rel(c)[scp(fv)] | c \in C, scp(fv) \subseteq scp(c)\}$, and remove invalid tuples from constraint relations after generating each compound factor variable. For example, for the CSP in Figure 6.1, we can remove the tuple $\{(x_1, 1), (x_2, 1), (x_5, 0), (x_6, 0)\}$ from $rel(c_2)$ after generating the factor variable fv_1 on $\{x_1, x_2\}$. As $\{(x_1, 1), (x_2, 1)\}$ is removed, then $\{(x_5, 0), (x_6, 0)\}$ is not included in the domain of the factor variable fv_3 on $\{x_5, x_6\}$.

(iii) Constructing a bipartite encoding instance based on the generated factor variables and partitions (Lines 7-13)—this is a straightforward construction using Definition 6.2.

The number of possible partitions of constraint scopes is exponential in the number of constraints and constraint arity. Thus, many BE instances can be constructed for a CSP by applying different partitions. Next we introduce a heuristic used to generate some disjoint partitions in the following subsection.

6.3.1 Maximum edge partition

We now give a method to partition the non-binary constraints taking into account the possibility of obtaining a feasible higher level consistency. We first introduce some notations. We use $scp(e) = scp(c_i) \cap scp(c_j)$ to denote the variables covered by an edge $e = \{c_i, c_j\}$ in the dual graph of a CSP (X, C). The size of e is the cardinality of $scp(e)$. The edge $e = \{c_i, c_j\}$ is a *maximum edge of* c_i in C if $e \subseteq C$ and the size of e is greater than 1 and largest on all edges in C including c_i, and e is a *maximum edge* in C if e is a maximum edge of c_i or c_j in C. A bipartite encoding $BE(P)$ covers an edge $e = \{c_i, c_j\}$ if there exists a factor variable fv in $scp(c_i^*) \cap scp(c_j^*)$ such that $scp(fv) = scp(e)$, e.g., the bipartite encoding given in Figure 6.1 covers the edge $\{c_1, c_2\}$, since $scp(c_1^*) \cap scp(c_2^*)$ includes a factor variable fv_1 on $\{x_1, x_2\}$. Our strategy is that for overall propagation efficiency, the size of the encoding should be compact, in particular, we will focus on bit representations used in GAC/AC propagators. We propose a bipartite encoding heuristic where the encoding is compact and can cover some maximum edges to obtain higher level consistency (see Proposition 6.2).

Algorithm 6.2 generates partitions of constraint scopes. The data structure $scpp[c_i]$ records the partition of $scp(c_i)$, and *cur* records a set of constraints which are not partitioned.

Algorithm 6.2: partition(C)

```
1  cur ← C
2  while cur ≠ ∅ do
3      E ← maxEdge(cur)
4      if E = ∅ then break
5      for {c_i, c_j} ∈ E do
6          S_k ← scp(c_i) ∩ scp(c_j)
7          scpp[c_i] ← {S_k, scp(c_i) \ S_k}
8          scpp[c_j] ← {S_k, scp(c_j) \ S_k}
9          cur ← cur \ {c_i, c_j}
10     end
11 end
12 for c_i ∈ cur do
13     Select a variable x from scp(c_i)
14     scpp[c_i] = {scp(c_i) \ {x}, {x}}
15 end
16 return scpp
```

At Line 3, Algorithm 6.3 is called to generate a subset E of maximum edges in cur (see next subsection for more details). If E is not empty, we use the maximum edges in E to partition the constraint scopes of the constraints connected by the maximum edges (Lines 5-8), otherwise we use a basic partitioning—for each constraint c_i in cur, we select the last variable x in $scp(c_i)$ and partition $scp(c_i)$ (between Lines 12 and 14).

Proposition 6.2. *Given any CSP $P = (X, C)$, if $BE(P)$ is AC and covers all edges whose sizes are greater than 1, then the CSP $P' = (X, C')$ is FPWC, where $C' = \{c'_i | c_i \in C\}$ and $scp(c'_i) = scp(c_i)$ and $rel(c'_i) = \{t(fv_j, a_1) \cup t(fv_k, a_2) \mid \{(fv_j, a_1), (fv_k, a_2)\} \in rel(c^*_i)\}$.*

Proof. For each constraints $c'_i, c'_j \in C'$ and a tuple $\tau \in rel(c'_i)$ such that $S = scp(c'_i) \cap scp(c'_j)$ includes at least 2 variables, τ has a valid support on c'_j, since $\tau[S]$ is in $\mathcal{D}(fv_k)$ and has valid supports on c^*_j, where $fv_k \in scp(c^*_i) \cap scp(c^*_j)$ is a factor variable on S. Recall, P' is AC (Proposition 6.1), so P' is FPWC (see Chapter 2 for more details about FPWC). □

Proposition 6.2 shows that enforcing AC on a BE instance $BE(P)$ can achieve FPWC on P', where P' has the same variables and constraints as P, except that some tuples in the constraint relations of $\mathcal{P}'$, which cannot be extended to a solution of P, may be deleted.

6.3.2 Maximum edge generation

In this subsection, we introduce how to generate a set E of maximum edges in a set cur of constraints such that:

- for each constraint c_i in cur, there is at most one partition of the constraint scope $scp(c_i)$ based on E, which means that each constraint in cur is included by at most one edge in E, i.e., $|\{scp(e) | e \in E, c_i \in e\}| \leq 1$ for each $c_i \in cur$;
- for each edge $e = \{c_i, c_j\}$ in E, the constraints c_i and c_j are size_splittable by $scp(e)$. A constraint c_k is *size_ splittable* by a variable subset $S_i \subset scp(c_k)$ or $S_j = scp(c_k) \setminus S_i$ if the size of the bit representation (see data structures used in the HTAC and CT algorithms) of c_k is greater than or equal to that of the corresponding constraints in the BE encoding, namely, the sum of the sizes of the binary constraints c^*_k and $\{c^x_l \in C^+ | scp(fv_l) \in \{S_i, S_j\}, x \in scp(fv_l)\}$.[1]

We propose to evaluate the size of the bit representations of the constraints in the original and bipartite instance as follows. We use $|rel(c_k)|(\sum_{x \in scp(c_k)} |\mathcal{D}(x)|)$ to evaluate the size of the bit representation of c_k, since a bit set with $|rel(c_k)|$ bits is used to record the supports

[1] We remark that experiments show HTAC on the HVE encoding is competitive with CT on the original CSP, and as the experiments show, we want to improve on both. This means that attention is also needed on the size of the bit representations of the constraints.

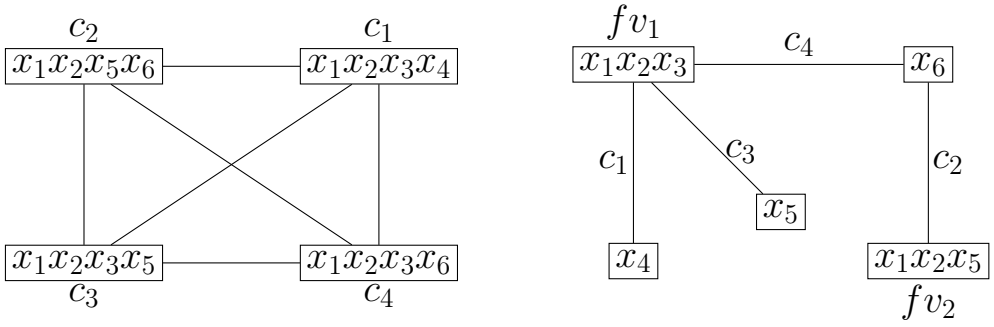
Algorithm 6.3: maxEdge(cur)

```
1  Let E be all maximum edges in cur
2  S ← {scp(c_i) ∩ scp(c_j) | {c_i, c_j} ∈ E}
3  for S_k ∈ S do
4      size[S_k] ← domain size of a factor variable on S_k
5      N[S_k] ← ⋃{e ∈ E | e ⊆ cur, scp(e) = S_k}
6  end
7  E_r ← ∅
8  while S ≠ ∅ do
9      Let M be the maximum cardinality subsets in S
10     Select S_k from M such that size[S_k]/|N[S_k]| is minimum
11     S ← S \ {S_k}
12     A ← {c ∈ (N[S_k] ∩ cur) | c is size_splittable by S_k}
13     E_A ← {e ∈ E | scp(e) = S_k, e ⊆ A}
14     E_r ← E_r ∪ E_A
15     cur ← cur \ (⋃_{e∈E_A} e)
16 end
17 return E_r
```

on c_k of each variable value in $\mathcal{D}(x)$. For each binary constraint between two variables v_1, v_2 in the BE, $|\mathcal{D}(v_1)| \times |\mathcal{D}(v_2)|$ is used to estimate the size of the constraint, since we represent binary constraints as binary matrices.

Algorithm 6.3 gives our maximum edge generation algorithm. The data structure $size[S_k]$ records the domain size of the compound factor variable fv_k on S_k (at Line 4). At Line 5, $N[S_k]$ is a set of constraints where the constraint scopes may be partitioned by S_k. At Lines 9 and 10, we first select a variable subset S_k from S with maximum cardinality, and further select those with minimum $\frac{size[S_k]}{|N[S_k]|}$. Lines 12-14 evaluate which constraints in $N[S_k]$ are size_splittable by S_k, recording the corresponding maximum edges in E_r. We remark that we also delete some invalid tuples when calculating the domain size of a factor variable fv, where a tuple is invalid and deleted if it does not support any value in the domain of fv which is generated during calculating the domain size of fv.

Example 6.2. *Let* $P = (\{x_1, \cdots, x_6\}, \{c_1, \cdots, c_4\})$ *be a CSP with domains* $\{0, 1\}$. c_1, c_2, c_3 *and* c_4 *are the linear equations:* $x_1+x_2+x_3+x_4=1$, $x_1+x_2+x_5+x_6=2$, $x_1+x_2+x_3+x_5=2$ *and* $x_1+x_2+x_3+x_6=1$. *Figure 6.2a is the dual graph of* P. *Figure 6.2b gives the partitions*



(a) Dual graph **(b)** Partitions

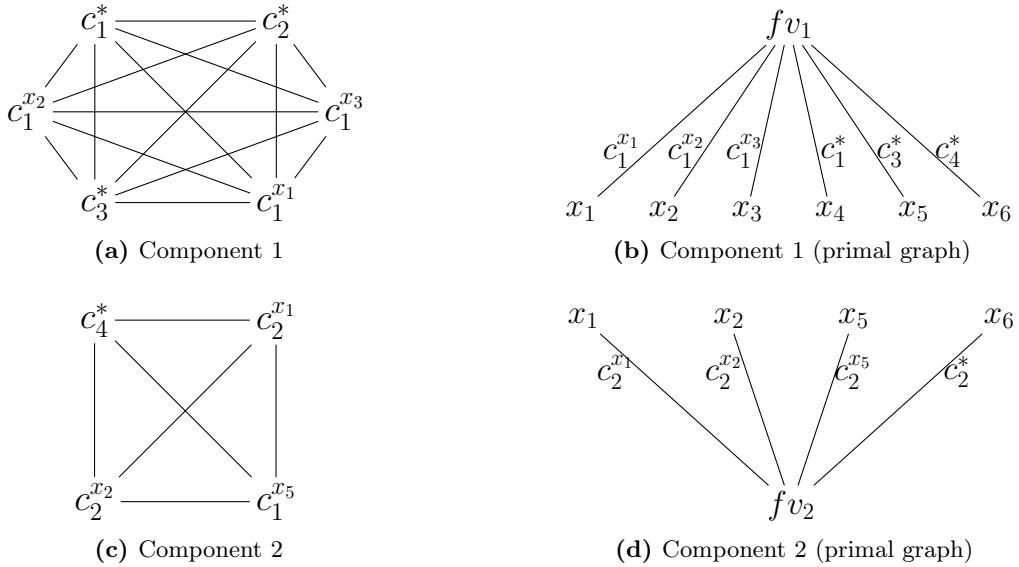
Figure 6.2: A maximum edge partition example.

generated by Algorithm 6.2. There are 5 maximum edges, the scopes are S_1=$\{x_1, x_2, x_3\}$, S_2=$\{x_1, x_2, x_5\}$ *and* S_3=$\{x_1, x_2, x_6\}$ *with* $N[S_1]$=$\{c_1, c_3, c_4\}$, $N[S_2]$=$\{c_2, c_3\}$ *and* $N[S_3]$=$\{c_2, c_4\}$. *We generate the factor variables on* S_1, S_2 *and* S_3, *and then record their domain sizes. At Line 10 in Algorithm 6.3,* S_1 *is selected.* c_1, c_3 *and* c_4 *are size_splittable by* S_1, *thus,* $\{c_1, c_3\}$, $\{c_1, c_4\}$ *and* $\{c_3, c_4\}$ *are the maximum edges selected. Then* $\{S_1, \{x_4\}\}$, $\{S_1, \{x_5\}\}$ *and* $\{S_1, \{x_6\}\}$ *are the partitions of* $scp(c_1)$, $scp(c_3)$ *and* $scp(c_4)$. *After partitioning the constraint scopes of* c_1, c_3 *and* c_4, *there are no more maximum edges, so we select* x_6 *from* $scp(c_2)$, *with partition of* $scp(c_2)$ *being* $\{\{x_1, x_2, x_5\}, \{x_6\}\}$. *Then we construct a BE instance* $(X^* \cup X, C^* \cup C^+)$ *such that* X^*=$\{fv_1, fv_2\}$, C^*=$\{c_1^*, c_2^*, c_3^*, c_4^*\}$, *and* C^+=$\{c_1^{x_1}, c_1^{x_2}, c_1^{x_3}, c_2^{x_1}, c_2^{x_2}, c_2^{x_5}\}$, *where* $scp(fv_1)$=S_1, $scp(fv_2)$=S_2, $scp(c_1^*)$=$\{fv_1, x_4\}$, $scp(c_2^*)$=$\{fv_2, x_6\}$, $scp(c_3^*)$=$\{fv_1, x_5\}$ *and* $scp(c_4^*)$=$\{fv_1, x_6\}$.

6.4 AC on Bipartite Encoding instances

HTAC (see Chapter 5) is a state-of-the-art specialized AC algorithm used to handle the non-binary constraints encoded by the HVE encoding. It exploits the special structure of the binary constraints arising from the HVE encoding—the subset of binary constraints in a HVE instance from encoding each non-binary constraint has a star structure. Correspondingly, a specialized propagator AC-H is used for each subset of binary constraints which has a star structural constraint graph. In addition, for the HVE instances, we only need to do search on the original variables, and search heuristics can apply the structure of the HVE instances. We refer readers to Chapter 5 for more details about the HTAC algorithm, e.g., the search heuristics and data structures used.

The bipartite encoding encodes non-binary constraints as a set of binary constraints so that any AC algorithm can be applied. However, in the same way that HTAC uses



(a) Component 1 **(b)** Component 1 (primal graph)

(c) Component 2 **(d)** Component 2 (primal graph)

Figure 6.3: Graphs of components.

a specialized propagator, we also use a specialized GAC algorithm called AC-BE, which exploits the structure of the binary constraints in a BE instance $(X \cup X^*, C^+ \cup C^*)$ extending the framework of HTAC. Especially, we split the binary constraints of the BE instance into a set of connected components in the undirected graph (C_1, E), which is a subgraph of the dual graph of the BE instance, and employ special propagators for the components, where $C_1 = C^+ \cup C^*$ and $E = \{\{c_i, c_j\} \subseteq C_1 | scp(c_i) \cap scp(c_j) \cap X^* \neq \emptyset\}$ and every component is treated as a set of binary constraints in C_1. In addition, we also only do search on the original variables in X (like the HVE instances).

Every connected component *com* in the graph (C_1, E) can be regarded as a single "special constraint", where the special constraint is GAC if all binary constraint in *com* are AC. Correspondingly, non-binary constraints are modelled as a set of special constraints such that each special constraint is consist of binary constraints. We highlight that in case some non-binary constraints in a CSP are not encoded as binary constraints, we can use GAC and AC propagators to respectively handle the non-binary constraints and the binary constraint components encoding the other constraints in the CSP.

Example 6.3. *There are 2 connected components for the BE instance given in Example 6.2. Figure 6.3a is the first component* $\{c_1^{x_1}, c_1^{x_2}, c_1^{x_3}, c_1^*, c_2^*, c_3^*\}$. *The constraint scopes of all constraints in the first component include the factor variable* fv_1. *Figure 6.3c shows the second component* $\{c_2^{x_1}, c_2^{x_2}, c_2^{x_5}, c_4^*\}$. *The compound factor variable* fv_2 *is included in the constraint scopes of all constraints in the second component. Note that the constraints in different components only share some original variables.*

Algorithm 6.4: AC-BE(com)

$V \leftarrow \bigcup\{scp(c)|c \in com\}$, $T \leftarrow \emptyset$, $U \leftarrow com$
while $\exists x \in V$ *s.t.* $|\{c \in U|x \in scp(c)\}| = 1$ **do**
  $T \leftarrow T \cup \{c \in U|x \in scp(c)\}$
  $U \leftarrow U \setminus \{c \in U|x \in scp(c)\}$
end
$V^T \leftarrow \bigcup\{scp(c)|c \in T\}$, $V^U \leftarrow \bigcup\{scp(c)|c \in U\}$
Let V^R be roots of the primal graph of T such that $V^T \cap V^U \subseteq V^R$
Let O be a topological order over V^T such that $\{O_1, \cdots, O_{|V^R|}\} = V^R$
for $i = |V^T|$ **to** $|V^R| + 1$ **do**
  Let c be the constraint in T such that $scp(c) = \{O_i, O_j\}$ and $i > j$
  if O_j *may not be AC on* c **then**
    enforce O_j is AC on c
    if $\mathcal{D}(O_j)$ *is empty* **then**
      return False
    end
  end
end
if $\neg AC(U)$ **then return** false
for $i = |V^R| + 1$ **to** $|V^T|$ **do**
  Let c be the constraint in T such that $scp(c) = \{O_i, O_j\}$ and $i > j$
  if O_i *may not be AC on* c **then**
    enforce O_i is AC on c
    if $\mathcal{D}(O_i)$ *is empty* **then return** False
  end
end
for each *original variable* $x \in V$ **do**
  if $\mathcal{D}(x)$ *is changed* **then**
    Add x to propagation queue
  end
end
return true

Comparing to the normal AC propagators liking AC3^{bit}, the specialized AC propagator for BE instances has a different propagation ordering which iteratively enforces GAC on a connected component of the graph (C_1, E) until all connected components of (C_1, E) are GAC. Algorithm 6.4 presents a specialized propagation algorithm for a component *com* which enforces AC on all binary constraints in *com* as follows. We first partition the binary constraints in *com* into two subsets T and U (between Lines 1 and 4) such that:

- the primal graph of T is acyclic,
- every node in the primal graph of U is included in at least one cycle, and
- every connected component in the primal graph of T has at most one node which is also included in the primal graph of U,

where the primal graph of a set of binary constraints C' is an undirected graph $(\bigcup_{c \in C'} scp(c), \{scp(c) | c \in C'\})$. If U is empty, then the primal graph of C is acyclic, otherwise we have a special "star structure" such that the primal graph of U is the "root" and the trees included in the primal graph of T are the "leaves".

While an AC propagator can be used on the entire component *com*, we use a more efficient approach recognizing the tree structure in T (following HTAC). At Lines 6-7, a subset V^R of V^T is set as roots of the primal graph of T, where V^T is the variables included by the primal graph of T, and V^R has the variables included by both the primal graphs of U and T, and each connected component of the primal graph of T includes exactly one node in V^R. The primal graph of T becomes a directed acyclic graph by setting the variables in V^R as roots, and we can construct a topological order O over V^T (at Line 8) such that $O_1, \cdots, O_{|V^R|}$ are the root nodes in V^R and for any $j > 1$ and $O_j \in V^T$, there is at most one constraint $c \in T$ such that $scp(c) = \{O_i, O_j\}$ and $i < j$.

For the binary constraints in T, we can call revise functions to update the variable domains from leaves to the roots (between Lines 9 and 16), and then from roots back to leaves (between Lines 19 and 24), where the revise functions are used to enforce a variable x is AC on a binary constraint c by removing inconsistent values from $\mathcal{D}(x)$. At Lines 11 and 21, a variable x may not be AC on a binary constraint c between the variables x and y if (i) it is the first time to enforce x is AC on c or (ii) the domain $\mathcal{D}(y)$ has been changed since the last time enforcing x is AC on c. Then at Lines 13 and 23, the propagator returns false if any variable domain becomes empty.

For the binary constraints in U, a normal AC propagator (Algorithm 6.5) is used to enforce AC. Between Lines 1 and 10 in Algorithm 6.5, a queue Q is used to record all variables which may cause reductions of another variable domain (same as the queue used in AC3^{bit}). If Q is not empty, then the algorithm iteratively propagates the variables in the queue, and calling revise functions to enforce AC (at Lines).

Algorithm 6.5: AC(U)

```
1  Q ← ∅
2  for c ∈ U do
3      {x, y} ← scp(c)
4      if y may not be AC on c then
5          Q ← Q ∪ {x}
6      end
7      if x may not be AC on c then
8          Q ← Q ∪ {y}
9      end
10 end
11 while Q ≠ ∅ do
12     pick and delete x from Q
13     for c ∈ U s.t. x ∈ scp(c) do
14         {x, y} ← scp(c)
15         if y may not be AC on c then
16             enforce y is AC on c
17             if D(y) is changed then
18                 if D(y) = ∅ then
19                     return False
20                 end
21                 Q ← Q ∪ {y}
22             end
23         end
24     end
25 end
26 return true
```

In addition, at Lines 12-26 in Algorithm 6.4, if the domain of a variable $x \in X$ is changed, then x is added to a propagation queue shared by all connected components of the graph (C_1, E), thus propagating to other constraints in C_1.

In our implementation, as common with the implementation of AC propagators, some revise functions are used. The revise functions are used to enforce a variable x is AC on a binary constraint between x and another variable y. Our revise functions are based on

those from HTAC (see Chapter 5) which use the following functions to update the variable domain $\mathcal{D}(x)$ based on another variable domain $\mathcal{D}(y)$: (i) *seekSupport* scans all values in the domain $\mathcal{D}(x)$, removing values which do not have any valid support on y; (ii) *reset* updates $\mathcal{D}(x)$ by using the union of all supports on x of the values in $\mathcal{D}(y)$; and (iii) *delete* removes all supports on x of the values which are removed from $\mathcal{D}(y)$. Based on our preliminary experiments, for each binary constraint c with a constraint scope $scp(c) = \{x, y\}$, we use the revise operations as follows. The *seekSupport* operation is used when x is an original variable or $|\mathcal{D}(x)| \leq |\mathcal{D}(y)|$. Otherwise, *reset* or *delete* operations are used to update the domain $\mathcal{D}(x)$. We use the *delete* operation as an optimization when the values in $\mathcal{D}(x)$ have at most 1 support in $\mathcal{D}(y)$. We adapt data structures employing ordered linked set [VHDT92, LS06], sparse set [BT93], bit set [SC06, LV08] and sparse bit set [DHL+16]to represent variable domains (see Chapter 2.5).

6.5 Experiments

Experiments presented in recent GAC algorithms (compact-MDD [VLS18], smart MDD [VLS19] and PW-CT [SC18] (enforces FPWC)) show CT to be state-of-the-art. HTAC is also shown to be comparable to CT, and better than STRbit. We compare our algorithm BE (AC on BE instances) with CT, STRbit and HTAC (AC on HVE instances). The experiments were run on a 3.20GHz Intel i7-8700 machine. All algorithms are implemented in the Abscon solver [MLB01].[2] We tested with the 3 well known variable search heuristics Wdeg/Dom (Wdeg) [BHLS04], Activity [MVH12] and Impact [Ref04] with the binary branching MAC and geometric restart strategy.[3] The value heuristic used is lexical value order. We measure CPU timings as: (i) solving time: time for solving the CSP; (ii) total time: initialization time (includes I/O, binary encoding) + solving time. Total CPU time is limited to 10 minutes per instance and memory to 8GB. We tested **all** 2559 non-binary instances (43 CSP series), which only employ table constraints, from the XCSP3 website http://xcsp.org:

> MaxCSP-{cnf-3-40, cnf-3-80, pedigree, spot5}, Kakuro-{easy, medium, hard}, Dubois, PigeonsPlus, Renault, Aim-{50, 100, 200}, Jnh, Cril, Tsp-{20, 25, 4-20}, Various, Nonogram, Bdd-{15, 18}-21, mdd-7-25-{5, 5-p7, 5-p9}, Reg2ext, Rand-3-24-{24, 24f}, Rand-3-28-{28, 28f}, Rand-5-12-{12, 12t}, Rand-5-{2, 4, 8}X, Rand-10-20-10, Rand-10-20-60, Rand-15-23-3, Rand-5-10-10, Rand-7-40-8t, Rand-8-20-5, Rand-3-20-{20, 20f}

We first evaluate the relative performance of the GAC/AC algorithmic on each variable

[2] https://www.cril.univ-artois.fr/~lecoutre/#/softwares

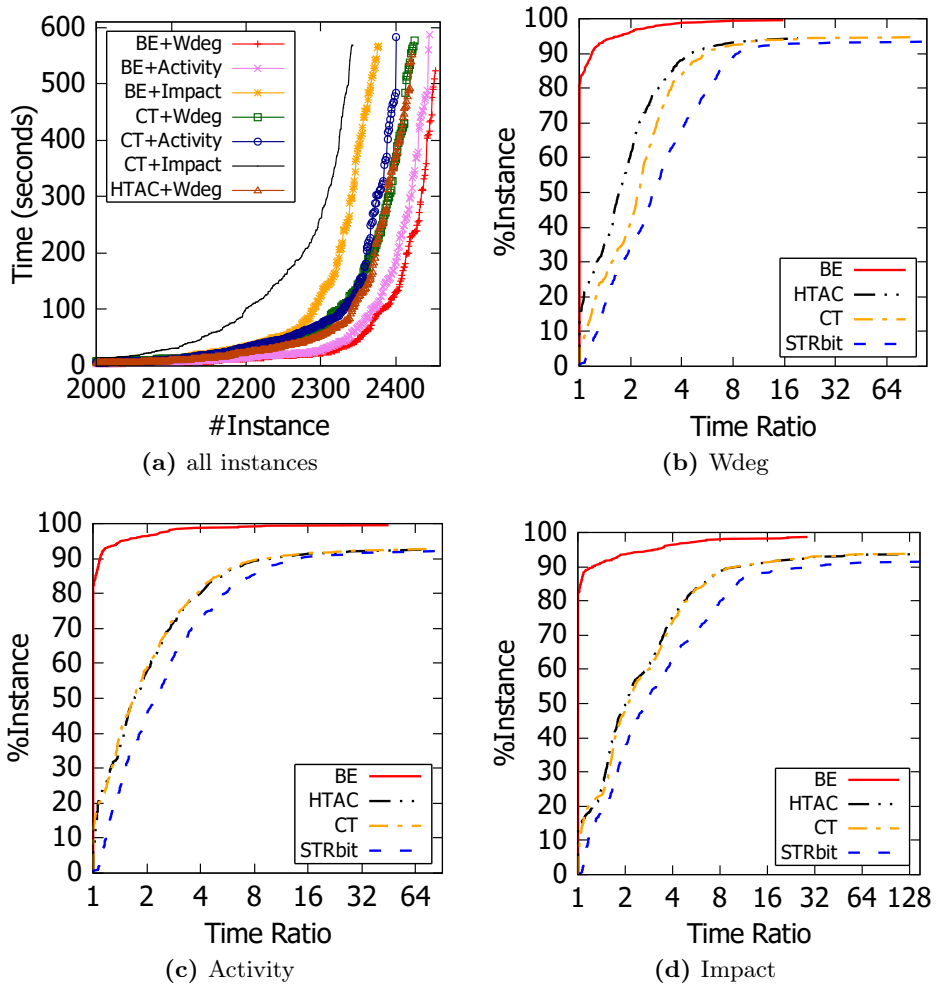
[3] The initial *cutoff* = 10 and $\rho = 1.1$. For each restart, *cutoff* is the allowed number of failed assignments and *cutoff* increases by (*cutoff* $\times \rho$) after restart.

		STRbit	CT	HTAC	BE
Wdeg (682)	AvgR	3.50	2.68	1.99	-
	MaxR	91.46	80.98	19.2	-
	#F	3	37	89	**553**
	#TO	45	36	38	**3**
	#BF	0	0	0	**148**
Activity (655)	AvgR	3.58	2.76	2.80	-
	MaxR	90.54	80.01	77.83	-
	#F	4	87	41	**539**
	#TO	51	47	48	**3**
	#BF	0	0	0	**137**
Impact (691)	AvgR	4.73	3.52	3.64	-
	MaxR	149.78	129.44	138.61	-
	#F	0	89	33	**569**
	#TO	59	44	43	**9**
	#BF	0	0	0	**175**
Initial Time (s)		1.39	1.37	1.38	1.65

Table 6.1: Relative comparison with total times, where "AvgR", "MaxR", "#F", "#TO" and "#BF" stand for "average time ratio", "maximum time ratio", "the number of fastest instances", "the number of timeout instances", "the number of backtrack free instances".

heuristic. To avoid timeout and small timings, for each variable heuristic, the trivial instances where all algorithms timeout or the solving time of the slowest algorithm is less than 2 seconds are ignored. Different search heuristics have a different number of non-trivial instances. Hence, the total number of instances in Table 6.1 varies for the experiments on each search heuristic. Table 6.1 gives a relative comparison between BE and other algorithms per search heuristic using *total time*: 682 instances for Wdeg; 655 for Activity; and 691 for Impact. AvgR is the average ratio of the total time of an algorithm to BE on the non-trivial instances. AvgR for Table 6.1 is greater than one as BE is faster on average than STRbit, CT and HTAC. We also show MaxR, the maximum total time ratio. BE can be much faster than STRbit/CT/HTAC. The average speedup of BE is between 1.99 to 4.7X and maximum speedup between 19 to 149X. The *Initial Time* row gives the average initialization time on all 2559 instances, and the initial time of BE is higher due to the time needed for encoding. #F (fastest) is the number of instances on which an algorithm has the smallest total time. The number of instances on which an algorithm is backtrack free is #BF and timeout is #TO. Overall BE solves more instances than CT, HTAC and STRbit, i.e., BE has fewer #TO. For example, by using the Activity variable search heuristic, BE is only timeout on 3 instances, while STRbit, CT and HTAC are timeout on 51, 47 and 48 instances. We highlight that BE is backtrack free on many instances, where BE has 137 or more backtrack free instances, showing that a higher consistency level is effective since GAC requires backtracking and search on these instances, i.e #BF = 0 for the GAC algorithms.

Figure 6.4a (in color) shows the runtime distribution of the *total time* for all 2559



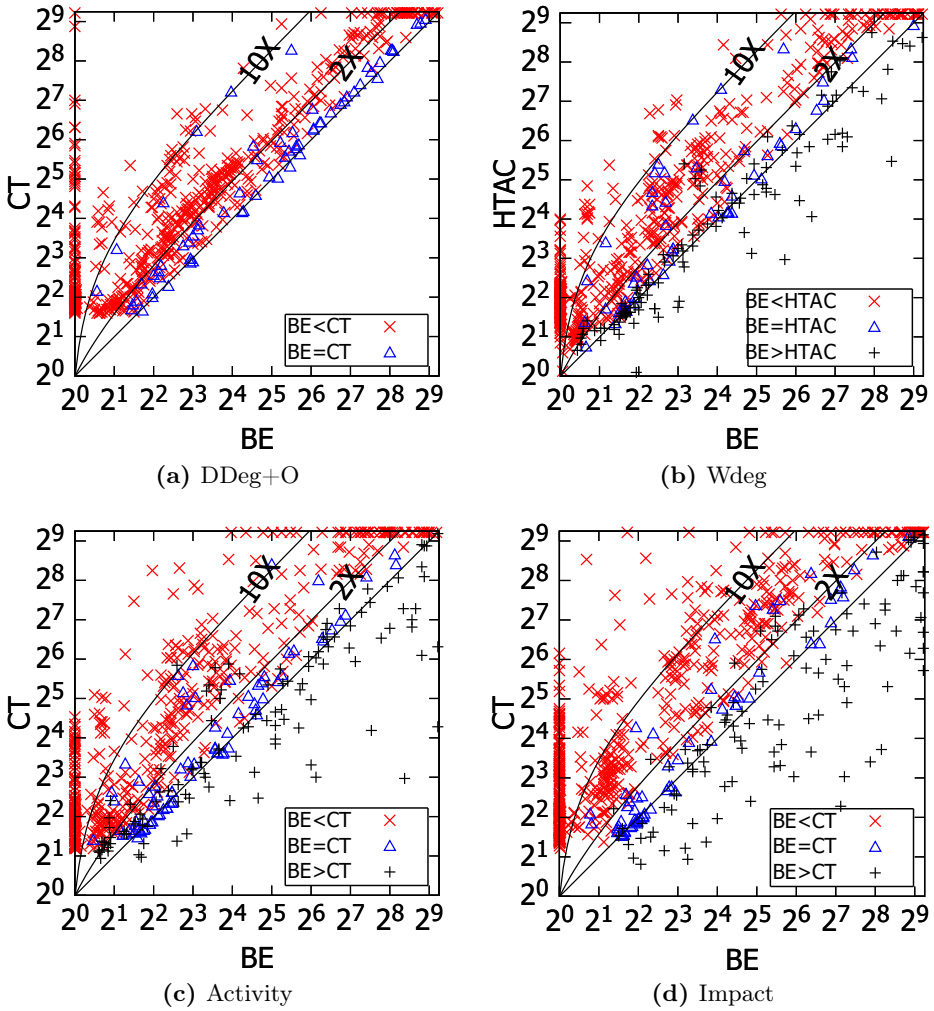
(a) all instances **(b)** Wdeg

(c) Activity **(d)** Impact

Figure 6.4: Overall *total time* comparison.

instances. It shows BE solves more instances than CT/HTAC with Wdeg and better than Impact and Activity. For example, BE solves 116 more instances than CT with Wdeg for a time limit of 30 seconds/instance. Figure 6.4b, 6.4c and 6.4d show the differences more clearly with performance profiles [DM02] comparing the algorithms for each search heuristic on the corresponding non-trivial instances. The y-axis is the percentage of instances and x-axis is the ratio of the total time of an algorithm to the virtual best algorithm (the best runtime per instance among all 4 tested algorithms). For example, when the x-coordinate is equal to 1, the y-coordinate is the percentage of instances on which an algorithm is the best algorithm. In Figures 6.4b,6.4c and 6.4d, BE is the best algorithm on more than 80% instances. BE dominates CT, HTAC and STRbit for all heuristics tested.

Figure 6.5 (in color) gives scatter plots comparing *only* solving times. The "△" ("×" and "+") dots in the plots denote the CSP instances on which the number of search nodes of



(a) DDeg+O **(b)** Wdeg

(c) Activity **(d)** Impact

Figure 6.5: *Solving time* comparison: Each dot denotes an instance, the time on the x-axis and y-axis is (1 + solving time) to enable logarithmic scales, "A=B" ("A>B" and "A<B") means the number of search nodes of Algorithm A is within 2% (greater than 1.02X and less than 1.02X) than that of B. The 10X (and 2X) line means BE is 10X (and 2X) faster than CT or HTAC.

BE is between 98% and 102% of (1.02X greater than and 1.02X less than) that of CT or HTAC. The solving time of BE is less than that of CT and HTAC on most instances, except instances where BE has the same or more search nodes. A few instances are slower—it turns out that the higher consistency level change the search tree, i.e., filtering impacts the search. To explore this effect, we tested the DDeg/Dom (DDeg) heuristic [SG97]. Figure 6.5a compares BE with CT using the static search heuristic DDeg+O, where the BE algorithm uses the structure of the original instances to calculate the value of "DDeg" [SG97], thus, BE and CT can have the same search tree. With the DDeg+O heuristic, the number of search nodes of BE is less than or equal to that of CT on all instances tested (583 non-trivial

instances for DDeg+O), confirming that AC on the BE encoding can be stronger than GAC on the original constraints. Additionally, the solving time of BE is less than that of CT on all but 10 (1.7%) "BE=CT" instances, moreover, BE is 10X (2X) faster than CT on 23% (71%) instances. Figures 6.5b, 6.5c and 6.5d give the results of Wdeg, Activity and Impact. For the heuristic Wdeg, Activity and Impact, the solving time of BE is 10X (2X) less than that of HTAC and CT on 27%, 31% and 35% (68%, 69% and 71%) instances, respectively. We see that the Impact heuristic is more affected by consistency levels. We remark that the backtrack free instances with BE can be seen as the points which are on the y-axis.

6.6 Conclusion

GAC algorithms have been the mainstay of CSP solvers for non-binary constraints. In Chapter 5, the well known hidden variable encoding was shown through the HTAC algorithm to be competitive with state-of-the-art table GAC algorithms such as CT on the original non-binary constraints. This is surprising since encoding non-binary constraints into binary constraints allowing the use of AC algorithms has been thought to be inferior to GAC on the original non-binary constraints. In this chapter, we show how to further improve the performance of binary encoding with a new binary encoding, the bipartite encoding (BE) based on partitioning constraint scopes. Ensuring AC on the binary constraints in BE encoded instances not only gives GAC on the original non-binary constraints but may also have higher consistency than GAC. We present an algorithm to construct BE instances with heuristics to keep the representation of the binary constraints compact while encouraging the possibility of higher consistency. We also give a new AC propagator, AC-BE, which exploits the structure of the binary constraints in BE instances working on connected components of the encoded constraint graph. Extensive experiments comparing solving CSPs with BE versus state-of-the-art GAC algorithms (CT, HTAC and STRbit) on the well known variable search heuristics (Wdeg, Activity, Impact) demonstrate the superiority of solving the CSPs with BE. Furthermore, the higher consistency which is achieved also contributes to faster solving and in some cases makes the CSPs backtrack free.

Part C

Decision Diagram and Automaton Constraints

Chapter 7

Binary Constraint Trees

7.1 Introduction

Many specific constraints which do not fit well with special purpose global constraints can be modelled with different ad-hoc constraints, such as Ordered Multi-valued Decision Diagram (MDD) constraints [CY10] and non-deterministic finite state automaton (NFA) constraints [QW06]. A number of Generalized Arc Consistency (GAC) algorithms using different representations have been proposed to enforce GAC on ad-hoc constraints (see the background of ad-hoc constraints given in Chapter 3).

Rather than using GAC propagators to enforce GAC on ad-hoc constraints, we can transform the ad-hoc constraints into binary constraints and use Arc Consistency (AC) algorithms to enforce AC on the binary constraints. Various binary encodings have been proposed to transform table constraints into binary constraints, such as Dual/Double Encoding [DP89, SW99b], Hidden Variable Encoding (HVE) [RPD90] and Bipartite Encoding (BE) (introduced in Chapter 6). However, there is a lack of binary encodings to encode the non-table ad-hoc constraints which can be more compact than table constraints. For example, we are not aware any existing binary encodings used to encode the decision diagram and automaton constraints, such as the MDD and NFA constraints, which can be exponentially smaller than table constraints. To this end, we will demonstrate a novel binary encoding to transform decision diagram and automaton constraints into binary constraints.

In this Chapter, we introduce a compact representation called Binary Constraint Tree (BCT) to model ad-hoc constraints, where the BCT is a set of binary constraints with a tree structure which extends the representation introduced in [Dec87, Dec90a] by allowing hidden variables. We prove that BCT constraints can be super-polynomially smaller than the MDD and NFA constraints on representing some families of constraints.[1] In addition, we give a binary encoding called direct tree binary encoding (DTBE) to transform various decision diagram and automaton constraints into BCT constraints. Moreover, we show that enforcing AC on the DTBE encoding can achieve GAC on the original constraints.[2] To the

[1] This essentially shows that BCT is more succinct than the MDD and NFA constraints.

[2] In Chapters 8 and 9, we will show that DTBE can be further reduced by use of reduction rules, and BCT

best of our knowledge, the DTBE encoding is the first binary encoding which can be used to transform decision diagram and automaton constraints into binary constraints.

7.2 Binary Constraint Trees

It is well-known that tree structured binary CSPs can be solved without backtrack by maintaining AC during search [Fre82]. If a tree structured binary CSP is AC, then all variable values can be extended to a solution of the binary CSP. We exploit this by observing that we can enforce GAC on a non-binary constraint by encoding it into a set of binary constraints with a tree structure, and then maintaining AC on the binary constraints. In this section, we show how to encode a non-binary constraint as a tree structured binary CSP called Binary Constraint Tree.

Definition 7.1. *A* Binary Constraint Tree *(BCT) is a normalized binary CSP whose constraint graph (primal graph) is a tree.*

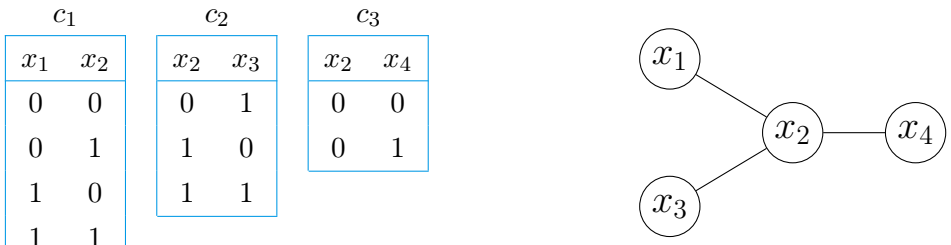
Dechter and Pearl [DP88] showed that any CSP instance can be transformed into a BCT by use of tree-clustering such that each solution of the BCT corresponds to a solution of the CSP. In this thesis, we show a BCT is more general and we represent a constraint as a BCT.

Definition 7.2. *A* BCT constraint c *is a pair* (V, P) *such that* $P = (X, C)$ *is a BCT and* $scp(c) = V$ *and* $V \subseteq X$ *and* $rel(c) = sol(X, C)[V]$*, where the variables in* $scp(c)$ *and* $X \setminus scp(c)$ *are called the* original *and* hidden *variables, respectively. A* tree binary encoding (TBE) *of a constraint* c^* *is a BCT* (X, C) *representing* c^**, where the BCT* (X, C) *represents* c^* *if* $scp(c^*) \subseteq X$ *and* $rel(c^*) = sol(X, C)[scp(c^*)]$.

By regarding different variables H included in a BCT (X, C) as hidden variables, the BCT can be used to represent different constraint relations $sol(X, C)[X \setminus H]$, where the set of other variables $X \setminus H$ in the BCT is the constraint scope (original variables). We remark that hidden variables can dramatically improve the expressive power of binary CSPs [Dec90b]. A large body of non-binary constraints cannot be directly represented as BCTs without using hidden variables. For example, a ternary clause constraint[3] c cannot be represented as a BCT without hidden variables, since removing a tuple from a binary constraint over two Boolean variables in $scp(c)$ delete 2 tuples over $scp(c)$, and there is only a tuple τ over $scp(c)$ such that $\tau \notin rel(c)$. Various properties of the BCTs without hidden variables have been discussed in [Dec87, Dec90a]. We will show that with hidden variables, BCTs can gain more expressive power, namely, BCT can be more compact than the MDD and NFA constraints.

AC propagators can outperform the state-of-the-art MDD and Table GAC propagators.

[3] A CNF clause over 3 Boolean variables.



(a) BCT

(b) Constraint graph

Figure 7.1: A BCT and its constraint graph.

Example 7.1. *Figure 7.1a is a BCT $P = (X, C)$ including 3 binary constraints $C = \{c_1, c_2, c_3\}$ over 4 variables $X = \{x_1, x_2, x_3, x_4\}$, where $scp(c_1) = \{x_1, x_2\}$ and $scp(c_2) = \{x_2, x_3\}$ and $scp(c_3) = \{x_2, x_4\}$ and variable domains are $\{0, 1\}$. The tree graph given in Figure 7.1b is the constraint graph (primal graph) of P, where each node and edge in the graph respectively denote a variable in X and a binary constraint in C. The BCT P can be used to represent the binary constraint c_3, since $scp(c_3) \subseteq X$ and $sol(P)[scp(c_3)] = rel(c_3)$. However, $sol(P)[scp(c_1)] \neq rel(c_1)$ and $sol(P)[scp(c_2)] \neq rel(c_2)$, thus, P is not a TBE of the constraints c_1, c_2. Then P can be regarded as a TBE of c_3 where $scp(c_3)$ is the set of original variables and x_1, x_3 are hidden variables.*

Given any BCT $P = (X, C)$ representing a constraint c^*, the constraint graph of P is a tree. Therefore, if the BCT P is AC, then every literal of the variables in X can be extended to at least a solution of P, which means that the constraint c^* is GAC. Correspondingly, we can get the following result.

Proposition 7.1. *Enforcing AC on $P = (X, C)$ achieves GAC on c^*.*

Proof. P is a normalized binary CSP such that $scp(c^*) \subseteq X$ and $|scp(c^*)| = r$ and $sol(P)[scp(c^*)] = rel(c^*)$. The constraint graph of P is a tree, thus, given any variable $x \in scp(c^*)$, we can have an order O over X such that $O_1 = x$ and for all $O_j \in scp(c^*)$, there is at most 1 variable $O_i \in N(O_j)$ with $i < j$.

Assume P is AC. For any variable value $a_1 \in \mathcal{D}(x)$, we can construct a series of tuples $\{\tau_1, ..., \tau_r\}$ such that $\tau_1 = \{(O_1, a_1)\}$ and for $j > 1$, $\tau_j = \tau_{j-1} \cup \{(O_j, a_j)\}$ such that $a_j \in \mathcal{D}(O_j)$ and $\{(a_i, O_i), (a_j, O_j)\} \in rel(c)$ if $c \in C$ and $scp(c) = \{O_i, O_j\}$ and $i < j$ and $(O_i, a_i) \in \tau_{j-1}$. τ_r is a solution of P, thus, $\tau_r[scp(c^*)]$ is a support of a_1 on c^*. So x is GAC on c^* for all $x \in scp(c^*)$, i.e., c^* is GAC. □

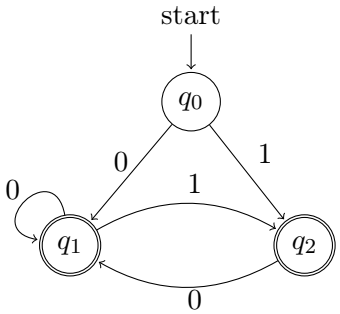
Based on Proposition 7.1, we can use AC propagators to maintain GAC on BCT constraints and the non-binary constraints represented by BCTs.

Example 7.2. *After enforcing AC on the 3 binary constraints of the BCT P given in Figure 7.1a, the domains of the variables x_1, x_2, x_3 and x_4 respectively become $\{0,1\}, \{0\}, \{0,1\}$ and $\{0,1\}$, where the variable value $(x_2, 1)$ is deleted due to it does not have any valid support on c_2. We can see that these variable values all can be extended to a solution of P, e.g., $(x_1, 0)$ is included by the solution $\{(x_1,0),(x_2,0),(x_3,0),(x_4,0)\}$. Therefore, the BCT P is GAC after enforcing AC on the binary constraints of P.*

7.3 Encoding decision diagram and automaton constraints

We then show how to encode the MDD [CY10] and NFA [QW06] constraints as BCT constraints. The definitions of the MDD and NFA constraints can be found in Chapter 3. We remark that the encodings of the MDD and NFA constraints can be directly used to handle the regular [Pes04] and Ordered Multi-valued Variable Diagram (MVD) [AFNP14, VLS18, VLS19] constraints, since each regular constraint (NFA constraint) can be expanded into a MDD (MVD) constraint, and every MDD (MVD) can be regarded as a DFA (NFA) [AFNP14], where the root and terminal nodes are respectively regarded as the initial and accepting states, and each edge corresponds to a transition.

Every regular constraint and NFA constraint can be directly decomposed into a set of Berge-acyclic ternary constraints [BCP04, QW06]. Therefore, a straightforward binary encoding of the regular and MDD constraints is that directly decomposing the decision diagram and automaton constraints into Berge-acyclic [BFMY83] ternary constraints, and then using Hidden Variable Encoding (HVE) to transform the ternary constraints into tree structured binary constraints. In the rest of this section, we introduce how to use this encoding to handle MDD and NFA constraints.



(a) DFA: nodes and edges respectively denote states and transitions

h_1	x_2	h_2
q_0	0	q_1
q_0	1	q_2

h_2	x_2	h_3
q_1	0	q_1
q_1	1	q_2
q_2	0	q_1

h_3	x_3	h_4
q_1	0	q_1
q_1	1	q_2
q_2	0	q_1

h_4	x_4	h_5
q_1	0	q_1
q_1	1	q_2
q_2	0	q_1

(b) Ternary constraints

Figure 7.2: A regular constraint and its ternary constraint decomposition.

Example 7.3. *Figure 7.2a is the DFA of a regular constraint over 4 variables X=$\{x_1, x_2, x_3, x_4\}$, where variable domains are $\{0, 1\}$, and the variable order is from x_1 to x_4. Every string accepted by the DFA with length 4 denotes a tuple in the constraint relation. Figure 7.2b gives the ternary constraints decomposing the regular constraint. The domains of the variables $\{h_1, \cdots h_5\}$ are the states in the DFA, where $\mathcal{D}(h_1)$ and $\mathcal{D}(h_5)$ respectively denote the initial and accepting states, and each tuple in the ternary constraint relations denotes a transition in the DFA.*

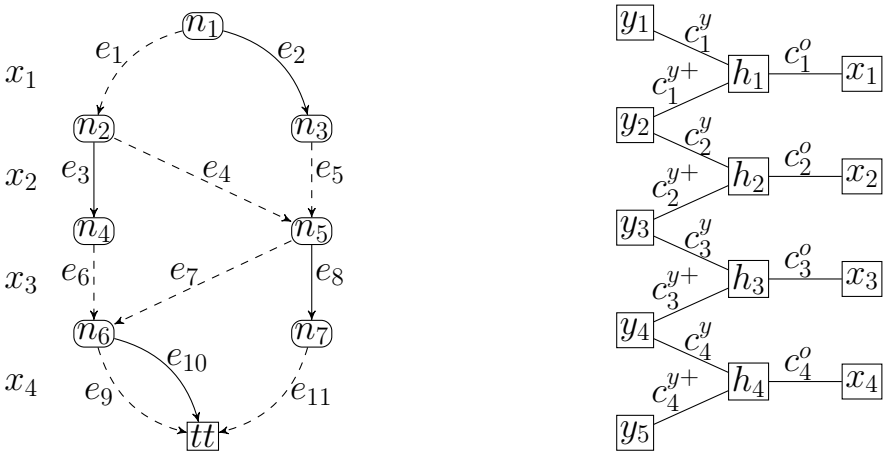
7.3.1 Direct tree binary encoding for MDD constraints

We now present a new binary encoding for MDD constraints, which directly encodes the nodes and edges in each layer of a MDD as hidden variables, and then uses binary constraints to connect the edges with the nodes and labels. The details of the encoding are given in the following definition.

Definition 7.3. *A* direct tree binary encoding *(DTBE) of a constraint c^* w.r.t. a MDD $mdd(c^*, O)$ is a BCT $dtbe(c^*) = (Y \cup H \cup scp(c^*), \{c_1^o, c_1^y, c_1^{y+}, ..., c_r^o, c_r^y, c_r^{y+}\})$ where*

- *$r = |scp(c^*)|$, $Y = \{y_1, ..., y_{r+1}\}$, $H = \{h_1, ..., h_r\}$, and O is an order over $scp(c^*)$;*
- *$scp(c_i^o) = \{O_i, h_i\}$, $scp(c_i^{y+}) = \{y_{i+1}, h_i\}$ and $scp(c_i^y) = \{y_i, h_i\}$;*
- *$\mathcal{D}(y_i)$ is the set of nodes in the i^{th} layer of $mdd(c^*, O)$;*
- *$\mathcal{D}(h_i)$ is the set of edges which point from the nodes in the i^{th} layer to the next layer;*
- *$rel(c_i^y) = \{\{(h_i, a), (y_i, b)\} | a \in \mathcal{D}(h_i), b = out(a)\}$;*
- *$rel(c_i^o) = \{\{(h_i, a), (O_i, l)\} | a \in \mathcal{D}(h_i), l = label(a)\}$;*
- *$rel(c_i^{y+}) = \{\{(h_i, a), (y_{i+1}, b)\} | a \in \mathcal{D}(h_i), b = in(a)\}$.*

DTBE transforms the ternary constraints decomposing a MDD into binary constraints with HVE, where each tuple in the hidden variable domains denotes an edge in the MDD. Figure 7.3a shows a MDD representing the regular constraint given in Example 7.3 with respect to the variable order from x_1 to x_4. The meaning of this constraint is that for any 2 consecutive variables, the value 1 is assigned to at most 1 variable. Figure 7.3b is the DTBE of a constraint c^* w.r.t. the MDD given in Figure 7.3a. The MDD has 4 layers of edges and 5 layers of nodes which correspond to the hidden variables $H = \{h_1, \ldots, h_4\}$ and $Y = \{y_1, \ldots, y_5\}$. The domains of the hidden variables in H and Y are the nodes and edges at each layer of the MDD, e.g., the hidden variable domain $\mathcal{D}(y_1) = \{n_1\}$ is the set of nodes in the first layer of the MDD and $\mathcal{D}(h_1) = \{e_1, e_2\}$ is the set of edges pointing from n_1. In Figure 7.3b, every node denotes a variable, and each edge corresponds to a binary constraint,



(a) MDD **(b)** DTBE

Figure 7.3: A MDD and the corresponding DTBE where the dashed (and solid) lines in the MDD denote value 0 (and 1).

where the constraint relations can be directly constructed using definition 7.3, e.g.,

$$rel(c_1^o)=\{\,\{(h_1,e_1),\,(x_1,0)\},\,\{(h_1,e_2),\,(x_1,1)\}\,\},$$
$$rel(c_1^y)=\{\,\{(h_1,e_1),\,(y_1,n_1)\},\,\{(h_1,e_2),\,(y_1,n_1)\}\,\},$$
$$rel(c_1^{y+})=\{\,\{(h_1,e_1),(y_2,n_2)\},\,\{(h_1,e_2),(y_2,n_3)\}\,\}.$$

We can see that the constraint graph of the DTBE encoding is a tree. In addition, every solution of $dtbe(c^*)$ denotes a path from the root to the terminal node in the MDD, which corresponds to a tuple in $rel(c^*)$, thus, $dtbe(c^*)$ is a BCT representing c^*. We also call $(scp(c^*), dtbe(c^*))$ as a DTBE constraint.

7.3.2 Direct tree binary encoding for NFA constraints

For any NFA constraint c^* over r variables, we can encode the states and transitions of the NFA into a sequence of hidden variables, and then representing the NFA constraint c^* as a BCT over the hidden variables such that every tuple in the constraint relation denotes a sequence of r transitions from the initial state to an accepting state in the NFA. The details of the encoding are given in Definition 7.4.

Definition 7.4. *A* direct tree binary encoding *(DTBE) of a NFA constraint* $c^* = (G, O)$ *is a BCT* $dtbe(c^*) = (Y \cup H \cup scp(c^*), \{c_1^o, c_1^y, c_1^{y+}, ..., c_r^o, c_r^y, c_r^{y+}\})$ *where*

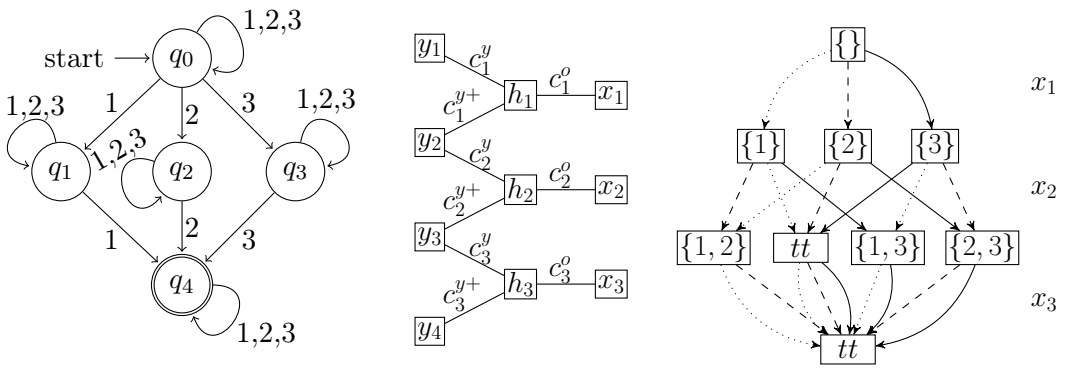
- $r = |scp(c^*)|$ *and* $G = (Q, \sum, \Delta, q_0, Q_t)$ *and* $Y = \{y_1, ..., y_{r+1}\}$ *and* $H = \{h_1, ..., h_r\}$*;*
- $scp(c_i^o) = \{O_i, h_i\}$*,* $scp(c_i^{y+}) = \{y_{i+1}, h_i\}$ *and* $scp(c_i^y) = \{y_i, h_i\}$*;*
- $\mathcal{D}(y_1) = \{q_0\}$*,* $\mathcal{D}(y_{r+1}) = Q_t$ *and* $\mathcal{D}(y_i) = Q$ *for* $2 \leq i \leq r$*;*

- $\mathcal{D}(h_i)$ *is the set of transitions in* G *for* $1 \leq i \leq r$*;*
- $rel(c_i^y) = \{\{(h_i, tr), (y_i, b)\} | tr \in \mathcal{D}(h_i)$*,* tr *is a transition from* $b\}$*;*
- $rel(c_i^o) = \{\{(h_i, tr), (O_i, s)\} | tr \in \mathcal{D}(h_i)$*,* s *is the symbol of the transition* $tr\}$*;*
- $rel(c_i^{y+}) = \{\{(h_i, tr), (y_{i+1}, b)\} | tr \in \mathcal{D}(h_i)$*,* tr *is a transition to* $b\}$*.*

Example 7.4. *Consider the constraint* $\bigvee_{i=1}^{r} \bigvee_{j=i+1}^{r} (x_i = x_j)$ *over* r *variables* $\{x_1, ..., x_r\}$ *with variable domain* $\{1, .., r\}$*, which expresses the negation of an* AllDifferent *constraint [Rég94]. The size of the negation of the MDDs (Ordered Multi-valued Decision Diagrams) representing AllDifferent constraints is exponential in* r *[AFNP14].*

We can use a NFA $(\{q_0, ..., q_{r+1}\}, \{1, ..., r\}, \Delta, q_0, \{q_{r+1}\})$ *to model the constraint such that* $\Delta(q_0, i) = \{q_0, q_i\}$*,* $\Delta(q_i, i) = \{q_i, q_{r+1}\}$*,* $\Delta(q_{r+1}, i) = \{q_{r+1}\}$ *and* $\Delta(q_j, i) = \{q_j\}$ *for* $1 \leq i \leq r$ *and* $j \neq 0, i, r+1$*. Figure 7.4a gives a NFA for* $r = 3$*, and Figure 7.4c is a MDD modelling the NFA, where every subset of the domain corresponds to a node in the MDD.*

Figure 7.4b gives the constraint graph of a DTBE for a NFA constraint (G, O), where G is given in Figure 7.4a and O is the variable order $x_1 \leq x_2 \leq x_3$. The states and transitions in the NFA are respectively encoded as the hidden variables $y_1, \cdots, y_4$ and $h_1, \cdots, h_3$, where the states and transitions are encoded as hidden variable values, i.e., $\mathcal{D}(y_1) = \{q_0\}$, $\mathcal{D}(y_2) = \mathcal{D}(y_3) = \{q_0, q_1, q_2, q_3, q_4\}$, $\mathcal{D}(y_4) = \{q_4\}$ and the domain of h_1, h_2, h_3 is the set of all transitions $\{(q_i, a, q_j) | q_j \in \Delta(q_i, a), 0 \leq i \leq 4, 1 \leq a \leq 3\}$ in the NFA. Then binary constraints are used to combine transitions with symbols and states. The binary constraint relations can be constructed based on Definition 7.4. For example, $rel(c_1^y) = \{\{(y_1, q_0), (h_1, (q_0, a, q_i))\} | i \in \{0, a\}, a \in \{1, 2, 3\}\}$, $rel(c_1^{y+}) = \{\{(y_2, q_i), (h_1, (q_0, a, q_i))\} | i \in \{0, a\}, a \in \{1, 2, 3\}\}$, $rel(c_1^o) = \{\{(x_1, a), (h_1, (q_0, a, q_i))\} | i \in \{0, a\}, a \in \{1, 2, 3\}\}$.



(a) NFA: nodes and edges respectively denote states and transitions

(b) BCT: nodes and edges are variables and constraints

(c) MDD: dotted (dashed and solid) lines denote the value 1 (2 and 3)

Figure 7.4: Different representations of a constraint over 3 variables $\{x_1, x_2, x_3\}$.

7.4 The succinctness of BCT constraints

The DTBE encoding can be used to transform any MDD or NFA constraint into a BCT constraint. In this section, we show that BCT can be super-polynomially smaller than MDD and NFA on representing some families of constraints. It is straightforward that BCT constraints are more succinct than MDD constraints, since NFA constraints can be encoded as BCT constraints, and NFA constraints are more succinct than MDD constraints. For example, NFA constraints can be exponentially smaller than MDD constraints on representing the negation of AllDifferent constraints (see Example 7.4).

Theorem 7.1. *BCT constraints can be exponentially smaller than MDD constraints on representing NFA constraints.*

Proof. The DTBE of a r arity NFA constraints has $3r+1$ variables and $3r$ binary constraints. The hidden variable domains include at most $max(sn, tn)$ values, where sn and tn are the number of states and transitions in the NFA. In addition, each binary constraint relation has tn tuples. So the size of the DTBE is polynomial in that of the NFA constraint.

The size of the negation of a MDD (Ordered Multi-valued Decision Diagrams) representing AllDifferent constraints is exponential in the constraint arity r [AFNP14]. Therefore, the number of nodes in the MDD which represents the family of NFA constraints given in Example 7.4 is exponential in r, where r is the constraint arity and the NFA of the constraints has $r+1$ states and $4r+r^2$ transitions. So BCT can be exponentially smaller than MDD on representing the NFA constraints. □

Theorem 7.1 shows that BCT constraints can be exponentially smaller than MDD constraints on representing the negation of AllDifferent constraints. We then introduce a family of constraints, defined with bounded treewidth CSPs, which can be represented as BCT constraints in polysize but not NFA constraints.

Definition 7.5. *The treewidth of a CSP (X, C) is bounded by k if there is a tree decomposition (T, L) of the CSP (X, C) where*

- *T is a tree and each node v in T is labelled with $L(v) \subseteq X$ such that $|L(v)| \leq k+1$,*
- *for each constraint $c \in C$, there is a node $v \in T$ such that $scp(c) \subseteq L(v)$, and*
- *for any $x \in X$, the nodes v in T with $x \in L(v)$ form a connected sub-tree of T.*

We say (X, C) can be encoded as a constraint c if $X = scp(c)$ and $rel(c) = sol(X, C)$.

Example 7.5. *A tree decomposition of the CSP $P = (\{x_1, x_2, x_3, x_4\}, \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_4, x_1 \neq x_4\})$ is $(\{v_1, v_2\}, \{e\})$, where $e = \{v_1, v_2\}$ and the labels are $L(v_1) = \{x_1, x_2, x_3\}$ and $L(v_2) = \{x_2, x_3, x_4\}$. Then $|L(v_1)| = |L(v_2)| \leq 3$, therefore, the treewidth*

of the CSP P is bounded by 2. Moreover, the CSP P can be encoded as a BCT $P_1 = (\{v_1, v_2\}, \{c_e\})$ such that $\mathcal{D}(v_1) = sol(P)[L(v_1)]$, $\mathcal{D}(v_2) = sol(P)[L(v_2)]$, $scp(c_e) = \{v_1, v_2\}$ and $rel(c_e) = \{(v_1, \tau[L(v_1)]), (v_2, \tau[L(v_2)]) \mid \tau \in sol(P)[L(v_1) \cup L(v_2)]\}$. Every solution t of P_1 corresponds to the solution $\cup_{(v,\tau)\in t}\tau$ of P.

Every CSP (X, C) with a treewidth bounded by a positive integer k has a tree decomposition (T, L) such that $|T| \leq |X|$ and $|L(v)| \leq k+1$ for all $v \in T$. Then the CSP (X, C) can be transformed into a tree structured binary CSP (H, C_H) such that the solution set $sol(X, C)$ is equal to $\{(\bigcup_{(h,\tau)\in t}\tau) | t \in sol(H, C_H)\}$ (see the tree clustering algorithm given in [DP89, Dec03]), where H denotes the set of nodes in T and $\mathcal{D}(h)$ is a set of assignments over the variables in $L(h)$, i.e., the domain size $|\mathcal{D}(h)|$ is at most d^{k+1}.

Proposition 7.2. *If the treewidth of a CSP $P = (X, C)$ is bounded by a positive integer k and the domain size of the variables in X is at most d, then the CSP P can be encoded as a BCT (V, C') such that $|V| \leq 2|X|$ and $|\mathcal{D}(v)| \leq d^{k+1}$ for all $v \in V$.*

Proof. As shown in [DP89], P can be transformed into a tree structured binary CSP $P_1=(H, C_H)$ such that $sol(P)=\{(\bigcup_{(h,\tau)\in t}\tau) | t \in sol(P_1)\}$ where $|H| \leq |X|$ and $\mathcal{D}(h)$ denotes a set of assignments over variables $L(h)$ and $|\mathcal{D}(h)| \leq d^{k+1}$. Then P_1 can be encoded as a BCT (V, C') where $V = X \cup H$ and $C' = C_H \cup \{c_x | x \in X\}$ and c_x is a binary constraint between x and any variable $h \in H$ such that $x \in L(h)$ and $rel(c_x) = \{\{(x, a), (h, \tau) | \tau \in \mathcal{D}(h), (x, a) \in \tau\}\}$. So P can be encoded as a BCT (V, C') such that $|V| \leq 2|X|$ and $|\mathcal{D}(v)| \leq d^{k+1}$ for all $v \in V$. □

Every MVD over Boolean variables is a special case of NROBP (nondeterministic read-once branching program), where a *NROBP* is defined as a directed acyclic graph with one root, one terminal node, and with some edges labelled by Boolean variable values such that there is no directed path having two edges labelled with values of the same variable [Raz16]. We remark that the NROBP allows unlabelled edges.

Proposition 7.3. *The CSPs P over Boolean variables X with a treewidth w such that $|X| \geq 2^w$ cannot be encoded as the NFA constraints whose size is polynomial in $|X|$.*

Proof. Every NFA constraint over Boolean variables can be expanded into a MVD [AFNP14] whose size is up to $|X|$ times larger than the NFA defining the NFA constraint. In addition, each MVD over Boolean variables is a special case of NROBP, and Theorem 5 in [Raz16] shows that the size of any NROBP encoding P is not polynomial in $|X|$. So there is not any polysize NFA constraints encoding P. □

Based on Propositions 7.2 and 7.3, we can directly get that BCT constraints can be super-polynomially smaller than NFA constraints.

Theorem 7.2. *BCT constraints can be super-polynomially smaller than NFA constraints.*

Proof. The CSPs P over Boolean variables X with a treewidth w such that $|X| \geq 2^w$ can be encoded as a BCT constraint (V, C) such that $|V| \leq 2|X|$ and $|\mathcal{D}(v)| \leq 2^{w+1} \leq 2|X|$ for all $v \in V$ (based on Proposition 7.2). In addition, Propositions 7.3 shows that the size of any NFA constraint encoding P is not polynomial in $|X|$. So BCT constraints can be super-polynomially smaller than NFA constraints. □

Given any BCT (V, C) encoding a constraint c over Boolean variables and a MVD m encoding $sol(V, C)$. We can construct a NROBP encoding c by removing hidden variable value labels from the edges in m, i.e., the edges with hidden variable value labels in the MVD become unlabelled edges. Then BCTs can be super-polynomially smaller than NROBPs on encoding the constraints over Boolean variables (see the proof of Theorem 7.2), so there is not polysize MVD encoding $sol(V, C)$. In addition, the treewidth of (V, C) is bounded by 1, therefore, we can also get that BCT constraints can be super-polynomially smaller than the NFA and MVD constraints on encoding the CSPs with a treewidth bounded by 1.

7.5 Conclusion

In this chapter, we present a new representation of ad-hoc constraints, called Binary Constraint Tree (BCT), which is a set of binary constraints with a tree structure. We show that any MDD and NFA constraints can be transformed into BCT constraints by use of the direct tree binary encoding (DTBE). In addition, a BCT is a set of binary constraints, therefore, the DTBE encoding is a binary encoding of decision diagram and automaton constraints. Furthermore, we prove that BCT constraints can be super-polynomially smaller than MDD and NFA constraints on representing some families of constraints.

Chapter 8

Reduction rules for BCT constraints

8.1 Introduction

Binary constraint tree (BCT) is a constraint representation which can be more compact than Ordered Multi-valued Decision Diagrams (MDDs) and non-deterministic finite state automatons (NFAs). Many ad-hoc constraints can be encoded as BCT constraints. For example, we can use the direct tree binary encoding (DTBE) to transform the MDD, regular and NFA constraints into BCT constraints (see Chapter 7). Note that the ad-hoc constraints which can be encoded as MDD and NFA constraints, such as the table constraints, can also be encoded as BCT constraints with the DTBE encoding.

We have proved that BCT constraints can be super-polynomially smaller than MDD, regular and NFA constraints on encoding some families of constraints (see Theorems 7.1 and 7.2).[1] However, the DTBE encoding has a similar size to the original constraints. How then to make the DTBE encoding smaller? We propose to simplify DTBE constraints by use of some reduction rules, where the BCT constraints generated from the DTBE encoding of MDD, regular and NFA constraints are also called *DTBE constraints* in this thesis.

We introduce four reduction rules to simplify BCT constraints by eliminating, merging and reconstructing hidden variables, where (i) the first and second reduction rules are respectively used to eliminate the hidden variables which are only included by one and two binary constraints in a BCT constraint, (ii) the third reduction rule is used to merge connected hidden variables, and (iii) the fourth reduction rule is used to reconstruct hidden variable domains. In addition, we prove that it is NP-hard to construct the optimal BCT constraints by using the four reduction rules to simplify BCT constraints. Then we propose a heuristic-based algorithm to reduce DTBE constraints with 4 reduction rules. Our experimental results on a large set of benchmarks show that the four reduction rules can significantly reduce the size of BCT constraints. The reduced BCT constraints can be up to 2166 times smaller than the original DTBE constraints.

[1] Regular constraints are special cases of NFA constraints and can be directly expanded into MDD constraints [AFNP14].

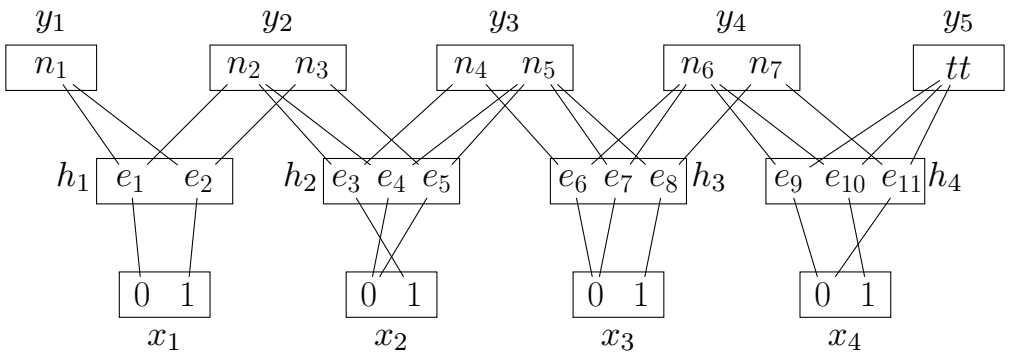


Figure 8.1: A DTBE of the MDD constraint given in Figure 7.3a, where every rectangle denotes a variable and the set of values in the rectangle is the domain of the variable, and each line between 2 values in two rectangles denotes a tuple in the relation of the constraint between the 2 variables corresponding to the two rectangles.

8.2 Reduction rules

In this section, we show how to make BCT constraints more compact. We propose four reduction rules to simplify any Tree Binary Encoding (TBE) $P_1 = (X_1, C_1)$ of a constraint c^*, where P_1 is a *TBE* of c^* if P_1 is a BCT and $scp(c^*) \subseteq X_1$ and $sol(P_1)[scp(c^*)] = rel(c^*)$. Figure 8.1 shows a DTBE of the MDD constraint given in Figure 7.3a, where the DTBE encoding is a special case of TBEs.

8.2.1 Two hidden variable elimination rules

The first rule (**Rule 1**) is to eliminate any hidden variable in P_1 which is only included by one constraint c in C_1. Assume $scp(c) = \{x, y\}$ and let $R_1(P_1, x) = (X_1 \setminus \{x\}, C_1 \setminus \{c\})$ be the CSP generated by eliminating x. Since c is removed, the domain of y is reconstructed as $\{b \in \mathcal{D}(y) | (y, b) \in rel(c_i)[\{y\}]\}$. A tuple over $X_1 \setminus \{x\}$ is a solution of $R_1(P_1, x)$ iff it can be extended to a solution of P_1, so $R_1(P_1, x)$ is a TBE of c^*.

Lemma 8.1. *$R_1(P_1, x)$ is a TBE of c^*.*

Proof. The constraint graph of $R_1(P_1, x)$ is a tree, since we can get a tree by eliminating a leaf from a tree. Given any tuple τ over the variables $X_1 \setminus \{x\}$.

If τ is a solution of $R_1(P_1, x)$, then $\tau[\{y\}]$ is in $rel(c)[\{y\}]$, which means there is $a \in \mathcal{D}(x)$ such that the tuple $\tau[\{y\}] \cup \{(x, a)\}$ is in $rel(c)$. Therefore, τ is in $sol(P_1)[X_1 \setminus \{x\}]$.

Conversely, if there is $a \in \mathcal{D}(x)$ such that $\tau \cup \{(x, a)\}$ is a solution of P_1, then τ is a solution of $R_1(P_1, x)$. Correspondingly, we can get that $sol(R_1(P_1, x))[scp(c^*)]$ is equal to $sol(P_1)[scp(c^*)]$ and $rel(c^*)$. So $R_1(P_1, x)$ is a TBE of c^*. □

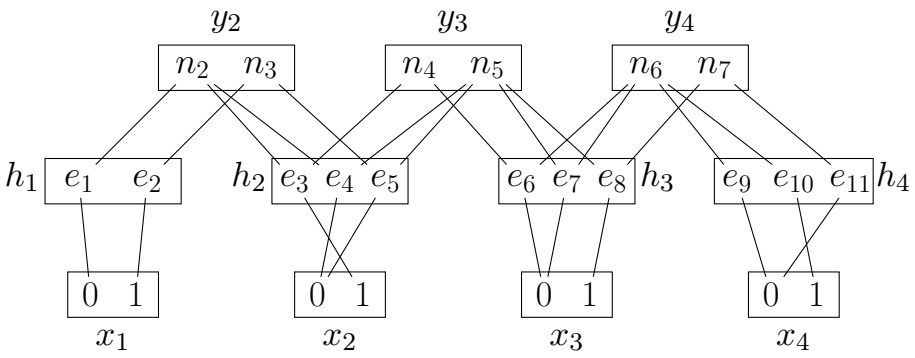


Figure 8.2: $P_{8.2}$: the TBE generated by using **Rule 1** on the DTBE given in Figure 8.1.

For the DTBE P given in Figure 8.1, the hidden variables y_1, y_5 are only included in one binary constraint, thus, they can be directly eliminated with **Rule 1**. Figure 8.2 shows the TBE $P_{8.2} = R_1(R_1(P, y_1), y_5)$ generated by applying **Rule 1** to eliminate y_1, y_5 from P. We can see that $P_{8.2}$ is a TBE of the DTBE constraint, and it has the same variables and binary constraints as P except that y_1, y_5 are eliminated.

Next, we present the second rule (**Rule 2**) that eliminates any hidden variable x which is only included by 2 constraints $c_i, c_j \in C_1$. Assume $scp(c_i) = \{x, y\}$ and $scp(c_j) = \{x, z\}$. Let $R_2(P_1, x) = (X_1 \setminus \{x\}, C \cup \{c'\})$ be the binary CSP generated by eliminating x, where $C = C_1 \setminus \{c_i, c_j\}$ and $scp(c') = \{y, z\}$ and $rel(c') = sol(\{x, y, z\}, \{c_i, c_j\})[\{y, z\}]$. Similarly, a tuple over $X_1 \setminus \{x\}$ is a solution of $R_2(P_1, x)$ iff it can be extended to a solution of P_1, thus, $R_2(P_1, x)$ is a TBE of c^*.

Lemma 8.2. *$R_2(P_1, x)$ is a TBE of c^*.*

Proof. The constraint graph of $R_2(P_1, x)$ is a tree, since the constraint graph is connected and the number of constraints (edge) equals to that of variables (nodes) minus 1. Then assume τ is a tuple over $X_1 \setminus \{x\}$.

If τ is a solution of $R_2(P_1, x)$, then τ is in $sol(X_1 \setminus \{x\}, C)$ and $\tau[\{y, z\}]$ is in $rel(c')$, which means that there is a value $a \in \mathcal{D}(x)$ such that the tuple $\tau[\{y, z\}] \cup \{(x, a)\}$ is in $sol(\{x, y, z\}, \{c_i, c_j\})$, thus, $\tau \cup \{(x, a)\}$ is a solution of P_1.

Conversely, if τ is in $sol(P_1)[X_1 \setminus \{x\}]$, then τ is included in $sol(X_1 \setminus \{x\}, C)$ and $\tau[\{y, z\}]$ is in $rel(c')$, thus, τ is also a solution of $R_2(P_1, x)$. Therefore, we can get that $sol(R_2(P_1, x))[scp(c^*)] = sol(P_1)[scp(c^*)]$.

In addition, P_1 is a TBE of c^* and $sol(P_1)[scp(c^*)] = rel(c^*)$, so $sol(R_2(P_1, x))[scp(c^*)] = rel(c^*)$ and $R_2(P_1, x)$ is a TBE of c^*. □

The hidden variables h_1, h_4, y_2, y_3, y_4 in Figure 8.2 can be eliminated by **Rule 2**. The TBE $P_{8.3} = R_2(R_2(R_2(R_2(R_2(P_{8.2}, h_1), h_4), y_2), y_4), y_3)$ is shown in Figure 8.3, where each 2

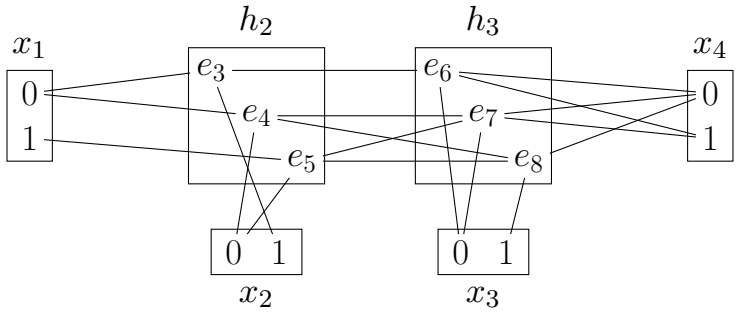


Figure 8.3: $P_{8.3}$: the TBE generated by using Rule 2 on the TBE given in Figure 8.2.

connected values in Figure 8.3 are also connected by some paths in Figure 8.2.

8.2.2 A hidden variable merging rule

The third rule (**Rule 3**) merges any 2 hidden variables x, y which are constrained by a constraint $c_i \in C_1$, i.e., $scp(c_i) = \{x, y\}$, as a new variable m such that $\mathcal{D}(m) = rel(c_i)$, while every constraint $c_j \in C_1$ between x (or y) and another variable $z \notin \{x, y\}$ is replaced with a new constraint c'_j where $scp(c'_j) = \{z, m\}$ and $rel(c'_j) = \{\{(z, a), (m, \tau)\} \mid \tau \in rel(c_i), \tau \cup \{(z, a)\}$ is in $sol(\{x, y, z\}, \{c_i, c_j\})\}$. We can get a new CSP $R_3(P_1, c_i) = (X \cup \{m\}, C \cup C')$ where $X = X_1 \setminus \{x, y\}$ and $C = \{c \in C_1 | x \notin scp(c), y \notin scp(c)\}$ and $C' = \{c'_j | c_j \in C_1, c_j \notin C, c_j \neq c_i\}$. For each tuple τ over X_1, τ is a solution of P_1 iff $\tau[X] \cup \{(m, \tau[\{x, y\}])\}$ is a solution of $R_3(P_1, c_i)$. So $R_3(P_1, c_i)$ is a TBE of c^*.

Lemma 8.3. *$R_3(P_1, c_i)$ is a TBE of c^*.*

Proof. The constraint graph of $R_3(P_1, c_i)$ is a tree. Assume τ is a tuple over X.

If there is $\tau' \in \mathcal{D}(m)$ such that $\tau \cup \{(m, \tau')\}$ is a solution of $R_3(P_1, c_i)$, then $\tau \in sol(X, C)$ and $\tau' \in rel(c_i)$ and for any $c \in C_1$ such that $c \notin C$ and $c \neq c_i$, $\{(m, \tau')\} \cup \tau[scp(c)]$ is in $rel(c')$ and $(\tau' \cup \tau)[scp(c)]$ is in $rel(c)$, thus, $\tau \cup \tau'$ is a solution of P_1.

Conversely, if there is $a \in \mathcal{D}(x)$ and $b \in \mathcal{D}(y)$ such that $\tau \cup \tau'$ is a solution of P_1 where $\tau' = \{(x, a), (y, b)\}$, then $\tau \in sol(X, C)$ and $\tau' \in \mathcal{D}(m)$ and for any $c \in C_1$ such that $c \notin C$ and $c \neq c_i$, $\tau' \cup \tau[scp(c)]$ is in $sol(scp(c) \cup \{x, y\}, \{c, c_i\})$ and $\{(m, \tau')\} \cup \tau[scp(c)]$ is in $rel(c')$, thus, $\tau \cup \{(m, \tau')\}$ is a solution of $R_3(P_1, c_i)$.

So we can get that $sol(R_3(P_1, c_i))[scp(c^*)]$ is equal to $sol(P_1)[scp(c^*)]$ and $rel(c^*)$, which means the binary CSP $R_3(P_1, c_i)$ is a TBE of c^*. □

Rule 3 encodes binary constraints as hidden variables by regarding constraint relations as variable domains, where each value in the domain corresponds to a tuple in the relation. We then present a rule to reconstruct hidden variables.

8.2.3 A hidden variable reconstruction rule

The fourth rule (**Rule 4**) reconstructs the domain of any hidden variable with its conditional c-tables defined as follows, where the definition of c-table can be found in Chapter 3.

Definition 8.1 (CC-T)**.** *Given a normalized binary CSP P and a variable x in P, the* conditional table *of x is a set of tuples $ct(P,x) = sol(N(x), NC(x))[N^-(x)]$. A* conditional c-table *(CC-T) of x is a c-table*[2] *representing $ct(P,x)$.*

We then show that the domain of any hidden variable in a BCT constraint can be reconstructed with a CC-T of the variable, correspondingly we can construct a smaller hidden variable domain by use of a smaller CC-T of the hidden variable.

For any variable x in the TBE P_1, the domain of x defines a CC-T $cct(P_1,x) = \{t^a \mid a \in \mathcal{D}(x)\}$ where $t^a = \{\ (y,b) \mid y \in scp(c),\ c \in NC(x),\ \{(y,b),(x,a)\} \in rel(c)\ \}$ is a c-tuple. For any constraint c in $NC(x)$, t^a includes all literals (y,b) such that $\{(y,b),(x,a)\} \in rel(c)$. Therefore, for each solution τ of P_1, if a literal (x,a) is in τ, then the tuple $\tau[N^-(x)]$ is a subset of t^a, otherwise there is a constraint $c \in NC(x)$ such that $\tau[scp(c)] \notin rel(c)$.

For example, $cct(P_{8.3}, h_3)$ is a CC-T including 3 c-tuples t^{e6}, t^{e7}, t^{e8}, where $t^{e6} = \{(h_2,e_3), (x_3,0), (x_4,0), (x_4,1)\}$, $t^{e7} = \{(h_2,e_4), (h_2,e_5), (x_3,0), (x_4,0), (x_4,1)\}$, and $t^{e8} = \{(h_2,e_4), (h_2,e_5), (x_3,1), (x_4,0)\}$. We can see that for any solution τ of $P_{8.3}$ including the literal (h_3,e_6), the tuple $\tau[\{h_2,x_3,x_4\}]$ is a subset of the c-tuple t^{e6}.

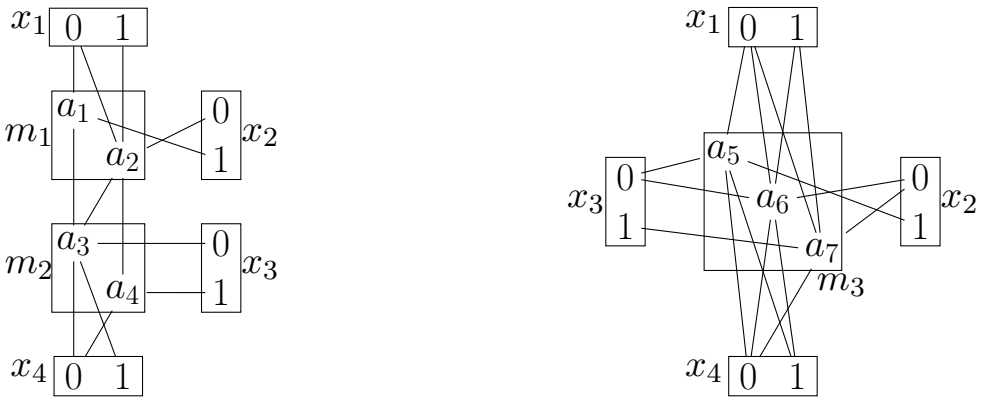
Conversely, given any CC-T rel of x, we can reconstruct x into a new variable x' such that $\mathcal{D}(x') = rel$ and replace each constraint c between x and another variable y with a new binary constraint c' such that $scp(c') = \{x',y\}$ and $rel(c') = \{\{(x',\tau'),(y,a)\} \mid \tau' \in \mathcal{D}(x'),(y,a) \in \tau'\}$. Correspondingly, we can get a new CSP $R_4(P_1,x,rel) = (X \cup \{x'\}, C \cup C')$ where $X = X_1 \setminus \{x\}$ and $C' = \{c' | c \in C_1, x \in scp(c)\}$ and $C = C_1 \setminus NC(x)$. The following lemma shows that $R_4(P_1,x,rel)$ is also a TBE.

Lemma 8.4. *$R_4(P_1,x,rel)$ is a TBE of c^*.*

Proof. P_1 is a BCT and **Rule 4** keeps the structure of P_1, i.e., the constraint graph of $R_4(P_1,x,rel)$ is connected and the number of constraints (edges) is equal to that of variables (nodes) minus 1. Therefore, $R_4(P_1,x,rel)$ is also a BCT. Given any tuple $\tau \in sol(X,C)$:

If there is a c-tuple $\tau' \in \mathcal{D}(x')$ such that the tuple $\tau \cup \{(x',\tau')\}$ is a solution of $R_4(P_1,x,rel)$, i.e., the tuple τ is in $sol(R_4(P_1,x,rel))[X]$, then the tuple $\tau[scp(c)] \cup \{(x',\tau')\}$ is in $rel(c')$ and $\tau[scp(c)] \subseteq \tau'$ for all $c \in NC(x)$, thus, $\tau[N^-(x)] \subseteq \tau'$. At the same time, $\mathcal{D}(x')$ is a c-table representing $ct(P_1,x)$, thus, $\tau[N^-(x)]$ is in $ct(P_1,x)$. $ct(P_1,x)$ is equal to $sol(N(x),NC(x))[N^-(x)]$, so τ is in $sol(P_1)[X]$.

[2]More details about c-table can be found in Chapter 3



(a) $P_{8.4a}$: Rule 4 on $P_{8.3}$ **(b)** $P_{8.4b}$: Rule 3 on $P_{8.4a}$

Figure 8.4: Applying **Rules 3,4** for reducing the TBE given in Figure 8.3.

Conversely, if τ is in $sol(P_1)[X]$, then $\tau[N^-(x)]$ is in $ct(P_1, x)$ and there is $\tau' \in \mathcal{D}(x')$ such that $\tau[N^-(x)] \subseteq \tau'$, since $\mathcal{D}(x')$ is a c-table representing $ct(P_1, x)$. Correspondingly, for any $c \in NC(x)$, $\tau[scp(c)] \cup \{(x', \tau')\}$ is in $rel(c')$, since $\tau[scp(c)] \subseteq \tau'$, so $\tau \cup \{(x', \tau')\}$ is a solution of $R_4(P_1, x, rel)$ and τ is in $sol(R_4(P_1, x, rel))[X]$.

So we can get that $sol(R_4(P_1, x, rel))[scp(c^*)]$ is equal to $sol(P_1)[scp(c^*)]$ and $rel(c^*)$, which means the BCT $R_4(P_1, x, rel)$ is a TBE of c^*. □

For the TBE $P_{8.3}$ given in Figure 8.3, the hidden variable h_2 can be reconstructed by **Rule 4** as a new hidden variable m_1 with the CC-T $rel_2 = \{a_1, a_2\}$, where $a_1 = \{(x_1, 0), (x_2, 1), (h_3, e_6)\}$ and $a_2 = \{(x_1, 0), (x_1, 1), (x_2, 0), (h_3, e_7), (h_3, e_8)\}$, and then the hidden variable h_3 can be reconstructed as a new hidden variables m_2 with the CC-T rel_3=$\{a_3, a_4\}$ where $a_3 = \{(x_3, 0), (x_4, 0), (x_4, 1), (m_1, a_1), (m_1, a_2)\}$ and $a_4 = \{(x_3, 1), (x_4, 0), (m_1, a_2)\}$. We show an example of **Rule 4** in Figure 8.4a which gives TBE $P_{8.4a} = R_4(R_4(P_{2b}, h_2, rel_2), h_3, rel_3)$. The constraint relations are reconstructed based on the CC-Ts, e.g., the relation of the constraint c_4^n between m_1 and m_2 is equal to $\{\{(m_1, a_1), (m_2, a_3)\}, \{(m_1, a_2), (m_2, a_3)\}, \{(m_1, a_2), (m_2, a_4)\}\}$, which is based on that $(m_1, a_1) \in a_3$, $(m_1, a_2) \in a_3$ and $(m_1, a_2) \in a_3$.

We can also merge the hidden variables m_1 and m_2 in P_{3a} as a new hidden variable m_3 with **Rule 3**. An example using **Rule 3** is Figure 8.4b giving the TBE $P_{8.4b} = R_3(P_{3a}, c_4^n)$, where each value in $\mathcal{D}(m_3) = \{a_5, a_6, a_7\}$ corresponds to a tuple in $rel(c_4^n)$. For example, a_5 corresponds to the tuple $\{(m_1, a_1), (m_2, a_3)\}$ and the values connecting to a_5 in Figure 8.4b also connect to a_1 or a_3 in Figure 8.4a.

8.2.4 Evaluating TBE sizes

Rules 1-4 can be used to make various new TBEs of a constraint. However, the new TBEs may not always be more compact than the original. Thus, we need to evaluate the sizes of the TBEs before reconstructing them. We do not use the rules which do not reduce the TBE as measured by the evaluated size. In this thesis, as we represent binary constraints as binary matrices, the size of any binary constraint c is evaluated as $|\mathcal{D}(x) \times \mathcal{D}(y)|$ where $scp(c) = \{x, y\}$. The evaluated size, $size(P)$, of any TBE P is the sum of the evaluated sizes of all binary constraints in P.

For example, the evaluated sizes of the TBEs given in Figures 8.1, 8.2, 8.3, 8.4a and 8.4b are 66, 56, 33, 20 and 24, respectively. We can see that $P_{8.4a}$ is the smallest TBE. The evaluated size of $P_{8.4a}$ is 3.3 times smaller than that of the original DTBE given in Figure 8.1. In addition, $size(P_{8.4b})$ is greater than $size(P_{8.4a})$, so we do not use **Rule 3** to merge the hidden variables m_1 and m_2 in $P_{8.4a}$.

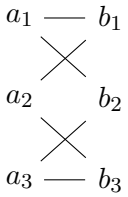
8.2.5 Constructing CC-Ts

We can reconstruct the domain of a hidden variable x in a TBE P with any CC-Ts of x via **Rule 4**. However, it is NP-hard to get the optimal c-table [KW07] representing the conditional table $ct(P, x)$. We construct a CC-T by merging some c-tuples in $ct(P, x)$ and $cct(P, x)$. Two c-tuples τ_1 and τ_2 in a CC-T rel_1 of x are *mergeable* w.r.t. a variable $y \in N^-(x)$ if $\tau_1 \setminus \tau_2$ or $\tau_2 \setminus \tau_1$ is a subset of the literals of y, since $\mathcal{T}(\tau_1) \cup \mathcal{T}(\tau_2) = \mathcal{T}(\tau_1 \cup \tau_2)$. The "mergeable" relation between the c-tuples in rel_1 is an equivalence relation which defines a partition $par(rel_1, y)$ of rel_1 where all c-tuples in the same subset S are mergeable w.r.t. y. We use $merge(rel_1, y) = \{\cup_{\tau \in S} \tau \mid S \in par(rel_1, y)\}$ to denote a c-table generated by merging c-tuples with $par(rel_1, y)$. Obviously, $merge(rel_1, y)$ is c-table representing $ct(P, x)$, thus, it is also a CC-T of x.

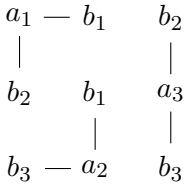
We then show how to generate a CC-T rel used for **Rule 4**. Let v be a variable in $N^-(x)$ with maximum domain size. If $\prod_{y \in (N^-(x) \setminus \{v\})} |\mathcal{D}(y)| < |\mathcal{D}(x)|$, which means that the number of c-tuples in $merge(ct(P, x), v)$ is less that of $cct(P, x)$, then the CC-T rel is initialized as $merge(ct(P, x), v)$, otherwise it is initialized as $cct(P, x)$. In addition, we go through the variables y in $N^-(x)$ one by one in an order of decreasing domain size and iteratively reset rel as the CC-T $merge(rel, y)$.

8.3 The complexity of simplifying BCT constraints

In this section, we prove that it is NP-hard to construct the optimal BCT by using the reduction rules to simplify DTBE constraints. The motivation is that the minimum number of bicliques which cover a bipartite graph can be generated from the optimal BCT representing



(a) Graph



(b) Bicliques

$(x_U, a_1), (x_V, b_1), (x_V, b_2)$
$(x_U, a_2), (x_V, b_1), (x_V, b_3)$
$(x_U, a_3), (x_V, b_2), (x_V, b_3)$

(c) c-table

Figure 8.5: A bipartite graph and a set of bicliques which cover the bipartite graph and a c-table corresponding to the set of bicliques.

a binary constraint, where the minimum biclique cover problem is NP-complete [Orl77], and the optimal BCT can be constructed by simplifying the DTBE constraint.

For any bipartite graph $G = (U \cup V, E)$, we can construct a binary constraint c_G over 2 variables x_U and y_V such that $\mathcal{D}(x_U) = U \cup U'$ and $\mathcal{D}(x_V) = V \cup V'$ and $rel(c_G) = \{\{(x_U, a), (x_V, b)\} | \{a, b\} \in E\}$ where $U' \cap U = \emptyset$, $|U'| = 9|U|$ and $V' \cap V = \emptyset$, and $|V'| = 9|V|$. Each c-table representing $rel(c_G)$ corresponds to a set of bicliques covering G, and vice versa. In addition, we will show that the CC-T of a hidden variable in the optimal BCT representing c_G is a minimum c-table representing $rel(c_G)$. Correspondingly the minimum biclique cover problem can be solved by constructing the minimum BCT representing c_G.

Example 8.1. *Figure 8.5a is a bipartite graph which includes the vertices* $\{a_1, a_2, a_3\} \cup \{b_1, b_2, b_3\}$. *Figure 8.5b shows 3 bicliques which cover the graph, e.g., the biclique* $G_1 = (\{a_1, b_1, b_2\}, \{\{a_1, b_1\}, \{a_1, b_2\}\})$ *covers the edges* $\{a_1, b_1\}$ *and* $\{a_1, b_2\}$. *Figure 8.5c gives a c-table corresponding to the bicliques, where each c-tuple in the c-table denotes a biclique, e.g., the c-tuple* $\{(x_U, a_1), (x_V, b_1), (x_V, b_2)\}$ *denotes the biclique* G_1.

The main result is given in Theorem 8.1, which shows that it is NP-hard to construct the optimal BCT by using the reduction rules to simplify the DTBE constraint encoding c_G. We then introduce the lemmas used to prove Theorem 8.1.

Lemma 8.5. *The optimal BCT representing* c_G *includes at least one hidden variable.*

Proof. Without loss of generality, assume $|U| \leq |V|$. We can construct a TBE $P = (\{x_U, x_V, x\}, \{x_U = x, c\})$ of c_G such that $\mathcal{D}(x) = U$ and $scp(c) = \{x, x_V\}$ and $rel(c) = \{(x, a), (x_V, b) | \{a, b\} \in E\}$, where x is a hidden variable.

Then the evaluated size of a BCT without hidden variables is equal to $100|U| \times |V|$ which is greater than $size(P) = 10|U| \times |V| + 10|U|^2$. So the optimal BCT representing c_G includes at least one hidden variable. □

From Lemma 8.5, we can get that the BCT which only includes the original variables x_U, x_V and the binary constraint c_G is not the optimal BCT representing c_G.

Lemma 8.6. *The optimal BCT $P = (X, C)$ representing c_G includes exactly one hidden variable h and $N^-(h) = scp(c_G)$.*

Proof. The hidden variable leaves can be eliminated with **Rule 1**, thus, P does not have any hidden variable leaves. In addition, P has at least one hidden variable (see Lemma 8.5), thus, the original variables x_U and x_V must connect to hidden variables.

The sum of the degrees of all $|X|$ variables is $2(|X|-1)$ and the sum of the degrees of the $|X|-2$ hidden variables is at least $2|X|-4$, which means the sum of the degrees of the 2 original variables is at most 2. Therefore, the 2 original variables in $scp(c_G)$ must be leaves and the degrees of the hidden variables are equal to 2.

Assume h is the hidden variable in P with the minimum domain size and P_1 is the TBE of c_G generated by using **Rule 2** to eliminate all hidden variables except h. If P has more than 1 hidden variable, then $size(P) > |\mathcal{D}(x_U)| \times |\mathcal{D}(h)| + |\mathcal{D}(x_V)| \times |\mathcal{D}(h)| = size(P_1)$. So the optimal BCT only includes one hidden variable h and $N^-(h) = scp(c)$. □

Lemma 8.7. *If P is an optimal BCT representing c_G and h is the hidden variable in P, then each minimum CC-T of h corresponds to a minimum biclique cover of G.*

Proof. For any c-table rel representing $rel(c_G)$, $\{G_\tau | \tau \in rel\}$ is a set of bicliques covering G, where $G_\tau = (U_\tau \cup V_\tau, E_\tau)$ is a biclique such that $U_\tau = \{a|(x_U, a) \in \tau\}$, $V_\tau = \{b|(x_V, b) \in \tau\}$ and $E_\tau = \{\{a, b\}|\{(x_U, a), (x_V, b)\} \in \mathcal{T}(\tau)\}$.

Conversely, for any biclique $G_1 = (U_1 \cup V_1, E_1)$ on G, $\tau_{G_1} = \{(x_U, a)|a \in U_1\} \cup \{(x_V, b)|b \in V_1\}$ is a c-tuple over $scp(c_G)$ such that $\mathcal{T}(\tau_{G_1}) \subseteq rel(c_G)$. Then for any set S of bicliques covering G, $\{\tau_{G_1}|G_1 \in S\}$ is a c-table representing $rel(c_G)$.

Based on Lemma 8.6, $N^-(h) = scp(c_G)$ and $ct(P, h) = rel(c_G)$, thus, each minimum CC-T of h is a minimum c-table representing $rel(c_G)$, which means each minimum CC-T of h corresponds to a minimum biclique cover of G. □

Lemma 8.7 shows that a minimum biclique cover of G can be generated from the CC-T defined by the domain of the hidden variable in the optimal BCT representing c_G. Therefore, it is NP-hard to construct the optimal BCT.

Theorem 8.1. *It is NP-hard to construct the optimal BCT by using the reduction rules to simplify DTBE constraints.*

Proof. An optimal BCT representing c_G can be constructed by (i) eliminating hidden variables from the DTBE of any MDD representing c_G with **Rules 1,2** to generate a BCT which only includes one hidden variable h such that $N^-(h) = scp(c_G)$, and then (ii) using **Rules 4** to reconstruct the domain of h with an optimal CC-T of h.

A minimum biclique cover of G can be generated by constructing the optimal BCT representing c_G (see Lemma 8.7), so it is NP-hard to construct the optimal BCT by using the reduction rules to simplify DTBE constraints. □

Algorithm 8.1: ReducingDTBE(c^*)

1 $(X, C) \leftarrow dtbe(c^*)$
2 $ApplyRule1(X, C)$
3 $ApplyRule2(X, C)$
4 $ApplyRule4(X, C)$
5 $ApplyRule2(X, C)$
6 $ApplyRule3(X, C)$
7 $ApplyRule4(X, C)$
8 $ApplyRule2(X, C)$
9 **return** (X, C)

8.4 An algorithm for simplifying DTBE

Algorithm 8.1 is used to simplify a DTBE constraint c^* by applying the 4 reduction rules. It is NP-hard to construct the optimal TBE by simplifying DTBE constraints with the rules (see Theorem 8.1). As such, we propose the following heuristic based on preliminary experiments. We apply the reduction rules in a certain order and also process the hidden variables in an order O. Note that the rules are only applied when it can reduce the evaluated size of the DTBE. The algorithm first generates a DTBE (X, C) of the constraint c^* at Line 8.1. Then the following functions applying reduction rules are used in Algorithm 8.1

- $ApplyRule1(X, C)$ is called at Line 1. It scans all hidden variables $x \in X$ and applies **Rule 1** to eliminate x if x is only included by one constraint in C.
- $ApplyRule2(X, C)$ is called at Lines 2,4 and 7. It scans all hidden variables x and applies **Rule 2** to eliminate x if x is only included by two constraint c_i, c_j in C such that $|\mathcal{D}(y)| \times |\mathcal{D}(z)|$ is not greater than $|\mathcal{D}(x)| \times (|\mathcal{D}(y)| + |\mathcal{D}(z)|)$ where $scp(c_i) = \{x, y\}$ and $scp(c_j) = \{x, z\}$.
- $ApplyRule3(X, C)$ is called at Line 5. It repeatedly scans the hidden variables from O_1 to O_k and back to O_1 until **Rule 3** cannot be applied to merge any variables, where k is the number of hidden variables. For any hidden variable O_i, if there is a constraint $c \in C$ between O_i and another hidden variable O_j such that the evaluated size of $R_3((X, C), c)$ is less than or equal to that of the BCT (X, C), then O_i and O_j are merged by **Rule 3**.
- $ApplyRule4(X, C)$ is called at Lines 3 and 6. It scans all hidden variables in the order

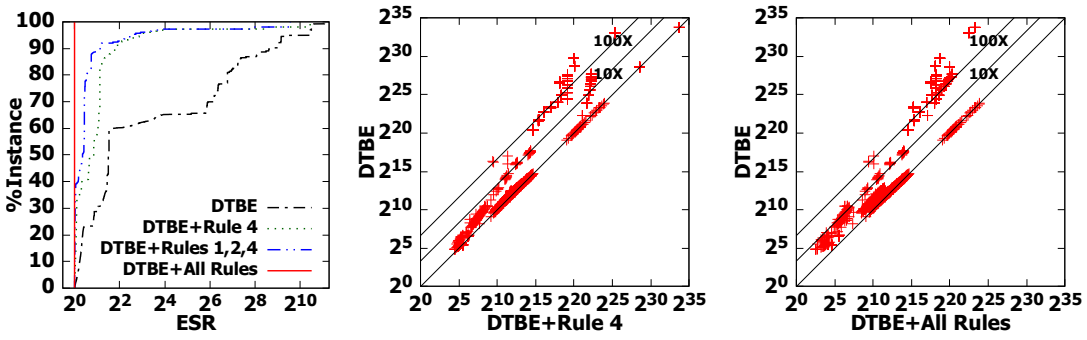
O and uses **Rule 4** to reconstruct the hidden variable domains. The detail of the CC-Ts used by **Rule 4** is given in the last subsection.

The initial order O over the hidden variables in $dtbe(c^*)$ is set as $y_{r+1} < h_r < y_r < ... < h_1 < y_1$ where $r = |scp(c^*)|$. After a new variable m is added by merging a variable x with its neighbors via **Rule 3** or reconstructing x via **Rule 4**, we update the order as $m < z$ (or $z < m$) for all variables z such that $x < z$ (or $z < x$).

8.5 Experiments

We compare the evaluated size of DTBE constraints with that of reduced BCT constraints. Experiments were run on a 3.20GHz Intel i7-8700 machine. Total memory is limited to 12G per instance. We use all 2559 non-binary instances which only employ table constraints from the XCSP website http://xcsp.org, where the table constraints are encoded as MDD constraints with the methods from [CY10, PR15].

Experimental results are shown in Figure 8.6. We compare the reduction rules with performance profiles [DM02] in Figures 8.6a. The various (colored) lines in Figure 8.6a "DTBE+*some Rules*" compare the BCTs generated with a different subset of rules in Algorithm 8.1, e.g., "DTBE+Rule 4" means only Rule 4 is used. Every dot (α, β) in Figure 8.6a denotes that there are β percentage instances such that the ESR (evaluated size ratio) is up to α, where ESR denotes the ratio of "the evaluated size of a BCT constraint" to "the evaluated size of the *virtual best BCT constraint*". The vertical red line (i.e., the "DTBE+All Rules" line) shows that the BCT constraints reduced by all 4 rules have the smallest size, of which ESR is equal to 1.0 on all instances. The more the reduction rules used, the smaller the BCT constraint constructed by reducing the DTBE constraints. The BCT constraints generated using all rules can be up to 2166 times smaller than the original DTBE constraints.



(a) Evaluated size **(b)** DTBE vs DTBE+Rule 4 **(c)** DTBE vs DTBE+ALL Rules

Figure 8.6: Results of reduction rules.

We then use scatter plots to compare the evaluated size of DTBE with DTBE+Rule 4 and DTBE+All. Each dot in Figures 8.6b and 8.6c denotes an instance, while the diagonal lines with the label 10X or 100X mean that the evaluated size of DTBE constraints is 10X or 100X larger than that of the reduced BCT constraints. The BCT constraints reduced with **Rule 4** (all rules) can be 10 and 100 times smaller the DTBE constraints on more than 31% and 13% (37% and 23%) instances, respectively. In addition, the average CPU time of Algorithm 8.1 applying all four reduction rules is only 0.43s.

8.6 Conclusion

BCT is an new interesting constraint as well as a binary encoding. It can be exponentially smaller than MDD constraints. However, the DTBE of MDD constraints may have the same size as the original constraints. In order to reduce the DTBE constraints, we give four reduction rules to reduce BCT constraint by eliminating, merging and reconstructing hidden variables. We prove that it is NP-hard to construct the optimal BCT constraints with the 4 reduction rules. As such we propose a heuristic-based algorithm to reduce DTBE constraints by using 4 reduction rules. Our experimental results on a large set of benchmarks show that the reduction rules can significantly reduce the size of BCT constraints. The reduced BCT constraints can be up to 2166 times smaller than the original DTBE constraints. Later, in Chapter 9, we will see that the AC propagator on reduced BCT constraints is very efficient.

Chapter 9

Enforcing AC on BCT constraints

9.1 Introduction

Ordered Multi-valued Decision Diagram (MDD) [SHMB90] is a compact representation which can be exponentially smaller than its corresponding table representation. Intuitively, Generalized Arc Consistency (GAC) propagators for MDD constraints have the potential to outperform those of table constraints as they exploit structures inside the constraints. However, prior works [VLS18, VLS19] show that table GAC algorithms, e.g., the CT algorithm, can outperform MDD GAC algorithms on a large set of benchmarks. As such an open question is "whether MDD GAC algorithms can overall outperform the CT algorithm [DHL+16]". We answer this by showing that solving table constraints as MDDs encoded with BCT constraints can outperform the state-of-the-art table GAC algorithm CT.

Instead of directly applying GAC algorithms, we can also use binary encodings to transform non-binary constraints into binary constraints, and then using Arc Consistency (AC) algorithms to enforce AC on the binary constraints. Especially, AC with binary encodings can achieve GAC or stronger consistencies on the original constraints [BSW08]. In addition, specialized AC algorithms with binary encodings can substantially outperform the state-of-the-art table GAC algorithms (see Chapter 6). In Chapters 7 and 8, we show that any MDD constraint can be encoded as a Binary Constraint Tree (BCT) constraint which is modelled with a set of binary constraints such that enforcing AC on the binary constraints can achieve GAC on the MDD constraint. BCT constraints can be more compact than MDD constraint. Therefore, we believe that it is also possible to improve the GAC algorithms of MDD constraints by transforming the MDD constraints into BCT constraints, and then using specialized AC propagators to enforce AC on the BCT constraints.

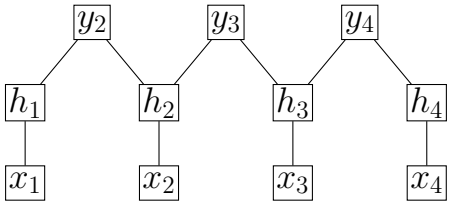
In this Chapter, we propose a specialized AC algorithm using bitwise operations to enforce AC on the BCT constraints encoding MDD constraints. Our experimental results on a large set of benchmarks show that the reduction rules given in Chapter 8 can significantly reduce the evaluated size of BCT constraints, correspondingly the AC propagator on BCT constraints can be much faster than the state-of-the-art MDD GAC propagator CD [VLS18] and also the table GAC propagator CT.

9.2 A BCT AC propagator

Recall from Chapter 7 that a BCT is a set of binary constraints with a tree structure, which we exploit to devise more efficient AC propagators. We follow the ideas of special propagation order used in the HTAC algorithm (Algorithm 5.1). The constraint graph of any Tree Binary Encoding (TBE) (X, C) of a constraint c^* is a tree. Hence, we can enforce AC on (X, C) by calling various revise functions to update the domains of variables from O_n to O_1 and back to O_n where $n = |X|$ and O is a *tree order* over X such that $|\{O_j \in N(O_i) | j < i\}| = 1$ for each $1 < i \leq n$. We use $preC(O_i)$ to denote the binary constraint between O_i and the variable $O_j \in N(O_i)$ such that $j < i$.

Algorithm 9.1 gives the details of the BCT AC propagator. It updates variable domains with a tree order from leaves to the root (between Lines 2 and 12), and then from the root back to leaves (between Lines 13 and 23). The propagator calls revise functions at Lines 5-11 and 16-22 to enforce that a variable x is AC on a binary constraint c between x and another variable y. Details about the revise functions and variable domain representations used in the propagator can be found in Sections 9.3 and 9.4. In addition, at Lines 5 and 16, the variable x may not be AC on c if it is the first time to enforce AC for x on c or the domain of y has been changed since the last time that AC has been enforced for x on c. This can be implemented by recording the size $lastsize[x, c]$ of $\mathcal{D}(y)$ or the times $lasttime[x, c]$ that $\mathcal{D}(y)$ has been updated when enforcing that x is AC on c. If $lastsize[x, c] > |\mathcal{D}(y)|$ or $lasttime[x, c]$ is less than the times that $\mathcal{D}(y)$ has been updated, then $\mathcal{D}(y)$ has been changed since the last time enforcing AC for x on c, therefore, x may not be AC on c.

Example 9.1. *Figure 9.1a gives the constraint graph of the BCT constraint (X, C) given in Figure 8.2. Each node and edge in the figure respectively denote a variable and binary constraint, where leaves are the original variables. One tree order O over X is $y_4 < y_3 < y_2 < h_4 < h_3 < h_2 < h_1 < x_4 < x_3 < x_2 < x_1$. For any $1 \leq i \leq 11$, there is at most a $1 \leq j < i$ such that $\{O_i, O_j\}$ is an edge in the figure.*



(a) Constraint graph

Ordered linked set	x_1, x_2, x_3, x_4
Bit set	x_1, x_2, x_3, x_4
Sparse set	y_2, y_3, y_4
Sparse bit set	h_1, h_2, h_3, h_4

(b) Domain representations

Figure 9.1: A constraint graph and domain representations.

Algorithm 9.1: AC-BCT(X, C)

```
1  Let O be a tree order over X
2  for i = n to 2 do
3      c ← preC(O_i)
4      {O_i, O_j} ← scp(c)
5      if O_j may not be AC on c then
6          enforce O_j is AC on c
7          if D(O_j) is changed then
8              if D(O_j) is empty then return False
9              if O_j is an original variable then add O_i to propagation queue
10         end
11     end
12 end
13 for i = 2 to n do
14     c ← preC(O_i)
15     {O_i, O_j} ← scp(c)
16     if O_i may not be AC on c then
17         enforce O_i is AC on c
18         if D(O_i) is changed then
19             if D(O_i) is empty then return False
20             if O_i is an original variable then add O_i to propagation queue
21         end
22     end
23 end
24 return True
```

9.3 Domain representations

We use four different variable domain representations in the BCT AC propagator, including ordered linked set [VHDT92, LS06], sparse set [BT93, dSMSSL13], bit set [SC06, LV08] and sparse bit set [DHL+16], where the bit set and ordered linked set are the default domain representations used in the Abscon solver. More details about the variable domain representations can be founded in Chapter 2.

For the original variables in the TBE (X, C) of a constraint c^*, we use the default variable

domain representations in the Abscon solver, i.e., both the ordered linked set and bit set, since the original variable domains can be shared with the GAC propagators of other constraints, e.g., various global constraints, which may use the default domain representations.

Then we use either sparse set or sparse bit set to represent hidden variable domains. We remark that these hidden variable domains are only used in the BCT AC propagator. Sparse bit set representations rather bit set are used because the hidden variable domains can be very large and bit set domains may have many ZERO words during search, and the sparse bit set need only keep the non-ZERO words. In addition, the BCT AC propagator does not require ordered hidden variable domains, thus, we can use the sparse set which is easier to be maintained during search than the ordered linked set [dSMSSL13].

In order to exploit efficient bitwise operations, we select a subset of hidden variables, $H_1 \subset X$, such that for each constraint $c \in C$, $scp(c) \cap H_1 \neq \emptyset$ or $scp(c) \subseteq scp(c^*)$, and representing the domains of these variable as sparse bit sets. In this way, every binary constraint in C has at least a variable of which the domain is represented as a bit set or sparse bit set, correspondingly efficient bitwise operations can be used to update variable domains. Then the domains of the remaining hidden variables, $H_2 = X \setminus H_1$, are represented as sparse sets. We do not represent all hidden variable domains as sparse bit sets, since our preliminary experiments show that it is better to use sparse sets for the variables in H_2. Note that we only use one kind of domain representations for each hidden variable to avoid the cost of maintaining consistency between different representations. For the TBEs used in this thesis, we set H_1 as $\{h \in X | h \notin scp(c^*), N(h) \cap scp(c^*) \neq \emptyset\}$.

Example 9.2. *Figure 9.1b gives the variable domain representations of the BCT constraint given in Figure 8.2. The original variables $x_1, \cdots, x_4$ are represented as both ordered linked sets and bit sets. The hidden variables $h_1, \cdots, h_4$, which can connect to at least an original variable, are represented as sparse bit sets. Then the hidden variables y_2, y_3, y_4, which do not connect to any original variable, are represented as sparse sets.*

9.4 Revise functions

We apply various variable domain representations in the BCT AC propagator, which leads to differences in the revise operations (revise functions) used to enforce AC on binary constraints. For the binary constraints including at least a variable in H_1, i.e., there is at least one variable whose domain is not represented as a sparse bit set, then we can directly use the revise functions introduced in Chapter 5, namely, the *seekSupport*, *reset* and *delete* functions. These revise functions are used to remove the values in $\mathcal{D}(x)$ which are not AC on a binary constraint c between 2 variables x and y. The *seekSupport* function is used when x is an original variable or $\mathcal{D}(x)$ is represented as a sparse set. Otherwise, the *reset* and *delete*

Algorithm 9.2: revise(c, x) with sparse bit set domains

```
1  Assume scp(c) = {x, y}
2  for xwi ∈ wordDom[x] do
3      xw ← bitDom[x][xwi]
4      for ywi ∈ wordDom[y] do
5          w ← 0
6          yw ← bitDom[y][ywi]
7          if bitcount(xw) < bitcount(yw) then
8              word ← xw
9              while word ≠ 0 do
10                 p ← numberTrailingZeros(word)
11                 word ← word & ¬(1 << p)
12                 bs ← bitSup[c, x, xwi, p]
13                 if (bs[ywi] & yw) ≠ 0 then   w ← w | (1 << p)
14             end
15         end
16         else
17             word ← yw
18             while word ≠ 0 do
19                 p ← numberTrailingZeros(word)
20                 word ← word & ¬(1 << p)
21                 w ← w | bitSup[c, y, ywi, p][xwi]
22             end
23         end
24         xw ← xw & ¬w
25         if xw = 0 then
26             break
27         end
28     end
29     bitDom[x][xwi] ← bitDom[x][xwi] & ¬xw
30     update wordDom[x]
31 end
32 return bitDom is changed
```

functions are used to update $\mathcal{D}(x)$, where the *delete* function can be used when all values in $\mathcal{D}(x)$ have at most one support in $\mathcal{D}(y)$.

In addition, for the binary constraints between 2 variables whose domains are represented as sparse bit sets, we apply a new revise function (Algorithm 9.2) for the case when both variables are represented as sparse bit sets. We first introduce the data structures used in our new revise function (see Algorithm 9.2). For any variable x:

- $bitDom[x]$ is a bit set representing $\mathcal{D}(x)$ and $bitDom[x][i]$ denotes the i^{th} word, where each '1' bit in the word corresponds to a value in $\mathcal{D}(x)$;
- $wordDom[x]$ is a sparse set recording the non-ZERO words in $bitDom[x]$;
- $bitSup[c, x, i, p]$ is a bit set recording all values in $\mathcal{D}(y)$ which supports the value corresponding to the p^{th} bit in $bitDom[x][i]$, where $scp(c) = \{x, y\}$ and a value $b \in \mathcal{D}(y)$ supports a value $a \in \mathcal{D}(x)$ if $\{(x, a), (y, b)\} \in rel(c)$.

The revise function (as usual) is used to enforce a variable x to be AC on a constraint c, i.e., eliminates values in $\mathcal{D}(x)$ which are not supported by any value in $\mathcal{D}(y)$, where $scp(c) = \{x, y\}$. The outer loop (Line 2), goes over each xwi in $wordDom[x]$ and a word xw records the values in $bitDom[x][xwi]$ whose supports in $\mathcal{D}(y)$ have not been found. The inner loop (Line 4), scans all ywi in $wordDom[y]$ and removes the values, which are supported by a value in the word $yw = bitDom[x][ywi]$, from xw. At Lines 28 and 29, the values in xw can be eliminated from $bitDom[x][xwi]$, since they have not any support in $\mathcal{D}(y)$. A main part of the function is checking which values in xw have supports in yw. If $bitcount(xw) < bitcount(yw)$ where $bitcount$ counts the number of '1' bits in a word, then the function scans all values in xw, using $numberTrailingZeros$ which return the position of the right most '1' bit in the word, and checks whether they have any supports in yw (between Lines 7 and 13), otherwise it scans all values in yw to compute a union w of the values in $bitDom[x][xwi]$ supported by the values (between Lines 16 and 20). At Line 23, w is used to update xw.

9.5 Experiments

We evaluate the efficiency of GAC algorithms in the Abscon solver.[1] We compare the propagators DTBE (AC on DTBE constraints) and TBE (AC on the BCT constraints generated by Algorithm 8.1) with CD (Compact-MDD), a state-of-the-art MDD GAC propagators also using bitwise operations [VLS18]. We tested the algorithms with the binary

[1] https://www.cril.univ-artois.fr/%7Elecoutre/#/softwares

branching MAC and geometric restart strategy.[2] The variable and value search heuristics used are Activity [MVH12] and lexical value order. Experiments were run on a 3.20GHz Intel i7-8700 machine. Total time is limited to 10 minutes per instance and memory to 12G, where total time is the sum of initialization time, transformation time and solving time. We tested with a set of (structured) benchmarks also used by other papers [CY10, GSS11, VLS18]:

- Car Sequencing: we use 30 Caroline Gagne hard instances [GGP05] (denoted as C-1) from CSPlib.[3] The problem is modelled with cardinality constraints and sequence constraints, where the sequence constraints are represented as Binary Decision Diagrams [CY10]. The MDDs used in C-1 are small. We create harder instances by increasing the block size and block capacity of the C-1 instances by 2 times (and 3 times) to create new instances with larger MDDs denoted as C-2 (and C-3).

- Pentominoes: we use 5 instances (denoted as P-m) from Minizinc Challenge 2020 and 36 instances from the pentominoes generator website.[4] These instances are modelled with regular constraints. We convert the regular constraints to MDD constraints. The instances are separated into 3 series P-10, P-15 and P-20 where P-10 (P-15 or P-20) includes the instances using a 10×10 (15×15 or 20×20) board and 10 (15 or 20) tiles.

- Nurse Scheduling: we use the model 1 and 2, denoted as N-1 and N-2, from [GSS11] where we only convert the regular constraints to MDD constraints and the global cardinality constraints are handled by the Abscon solver used. The MDDs used in N-1 and N-2 are small, so we create more challenging N-3 (N-4) instances with lager MDDs by changing the regular constraint of each nurse in N-2 to restrict that the nurse must work 1 or 2 night shifts every 9 (11) days, and 2 or 3 (3 or 4) days off every 7 (9) days, while the nurse can only work a second shift after 12 hours of the first. For each model, we use 50 instances from the N30 series[5] where the number of nurses in each instance is set to the maximum number of nurses required for any day. We remark that regular constraints are very natural in modelling scheduling problems.

- XCSP: we use all 2559 non-binary instances which only employ table constraints from the XCSP website[6] and the methods from [CY10, PR15] to encode tables into MDDs.

[2] The initial $cutoff = 10$ and $\rho = 1.1$. For each restart, $cutoff$ is the allowed number of failed assignments and $cutoff$ increases by ($cutoff \times \rho$) after restart.

[3] www.csplib.org

[4] https://github.com/zayenz/minizinc-pentominoes-generator

[5] https://www.projectmanagement.ugent.be/nsp.php

[6] http://xcsp.org

9.5.1 Non-table benchmarks

We first evaluate on problems with MDD constraints, the Car Sequence, Pentominoes and Nurse Scheduling Problems. The constraint relations of the MDD constraints used in these benchmarks are very large (see the number of edges #E in Table 9.1), and not practically representable as tables, hence, table GAC propagators are not evaluated. We use the MDDc propagator [CY10] as a baseline algorithm for these problems.

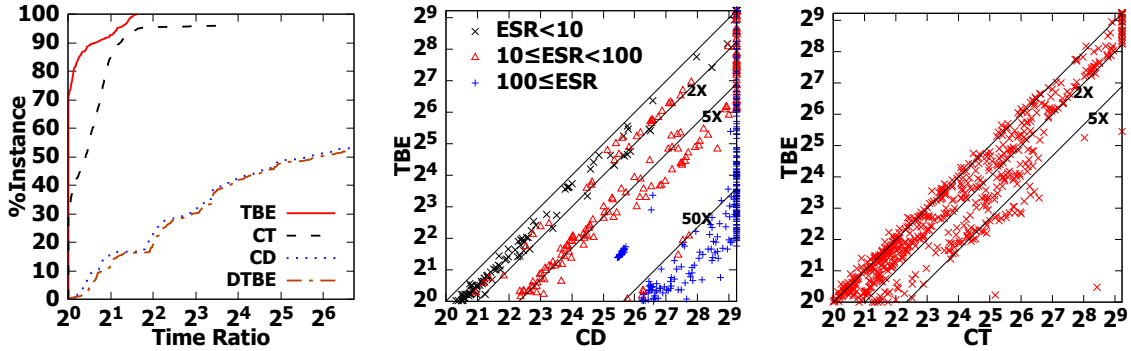
	#I	#N	#E	RT	Nodes/s				ESR
					TBE	DTBE	CD	MDDc	
C-1	30	1366	1903	0.03	**1308**	1198	1219	1253	3
C-2	30	16K	23K	0.13	**1730**	1478	1458	708	5
C-3	30	313K	435K	3.43	**814**	538	530	45	5
P-m	5	8366	95K	0.55	**5083**	2114	2423	602	46
P-10	12	7612	56K	0.28	**4392**	1070	1151	580	60
P-15	12	43K	511K	2.21	**1155**	75	81	46	162
P-20	12	137K	2M	11.13	**246**	14	15	4	347
N-1	50	1788	2319	0.01	**11512**	7818	7933	4211	13
N-2	50	1955	2914	0.01	**13174**	8897	9018	3384	26
N-3	50	13K	23K	0.06	**11618**	1827	1837	607	91
N-4	50	91K	161K	0.39	**3728**	44	43	96	255

Table 9.1: Non-table benchmarks. "#I", "#N" and "#E" stand for the number of "instances", the average number of "nodes" and "edges". "RT" and "ESR" stand for "average reduction time" and "mean evaluated size ratio".

The results are shown in Table 9.1. Many instances cannot be solved in 10 minutes, thus, we show the average number of search nodes per second, i.e., the number of search nodes divided by solving time. We highlight that these benchmarks are hard for the MDD solvers as the search speed can be as very low, e.g., 4 nodes/s (MDDc on P-20). The columns labelled #N and #E show the average number of nodes and edges of each MDD constraint. The ESR column is the mean ratio of "the evaluated size of DTBE constraints" to "the evaluated size of TBE constraints".[7] The RT column is the average CPU time (in seconds) of Algorithm 8.1.

We see the reduction rules work well on these benchmarks. The mean evaluated size of TBE is up to 347 times smaller than that of DTBE. More significantly, ESR increases as MDD constraints' sizes increase, so compression increases with larger MDDs. The DTBE propagator has similar performance to CD, showing that our basic representation is competitive. After rule reduction, the TBE propagator is much faster than other propagators on the problems tested. We can see that the larger the ESR, the faster the TBE propagator. For example, the mean evaluated size of TBE is 255 times (and 91 times) smaller than that of

[7] ESR is introduced in Section 8.5 to evaluate the reduction rules used in Algorithm 8.1.



(a) Total time **(b)** Solving time: CD vs TBE **(c)** Solving time: CT vs TBE

Figure 9.2: Results of table benchmarks.

DTBE on the N-4 series (and N-3 series), while TBE can be 85 times (and 6 times) faster than DTBE. In addition, the mean times of Algorithm 8.1 are acceptable. The transformation time is mostly small. The largest RT time in Table 9.1 is 11.13s, i.e., the average reduction time of P-20. This is not significant compared to the improvement, e.g., TBE runs at 246 search nodes/s on P-20 but MDDc only has 4 nodes/s and CD is 15 nodes/s.

9.5.2 Table benchmarks

We now evaluate table constraint benchmarks. The evaluation with non-table benchmarks was for ad-hoc constraints beyond the reach of tables. We now evaluate benchmarks when table or MDD constraints are both feasible representations.[8] We use CT [DHL+16] as the baseline algorithm as it is a state-of-the-art table GAC algorithm. CT has been shown to overall outperform MDD GAC algorithms on table constraint benchmarks (see the experimental results shown in [VLS18, VLS19]).

Experimental results are shown in Figure 9.2. We compare the AC/GAC propagators with performance profiles [DM02] in Figures 9.2a. Every dot (α, β) in Figure 9.2a denotes that there are β percentage instances such that the ratio of "the total time of a propagator" to "the total time of the *virtual best propagator*" is up to α. The total time includes the time of encoding table constraints as MDDs and BCTs. In Figure 9.2a, we remove: (i) the instances that cannot be solved by TBE, DTBE, CT or CD within 10 minutes (timeout); and (ii) the instances on which the solving times of CT and TBE are both less than 2 seconds (trivial). We use scatter plots to compare the solving time of TBE with CD and CT. Each dot in Figures 9.2b and 9.2c denotes an instance, while the diagonal lines with the label 2X,

[8] Recall that any ad-hoc constraint with table representation can be also represented as a MDD. In Section 9.5.1, the MDD constraints are simply too large to be represented with table constraints, but the table benchmarks are much smaller so either representation is feasible.

5X or 50X mean that the TBE algorithm is 2X, 5X or 50X faster than CD or CT.

Figure 9.2a with the time ratio shows clearly that the state-of-the-art MDD GAC algorithm CD on the table benchmarks is outperformed by the table GAC algorithm CT. The DTBE propagator is competitive with CD, while the TBE propagator is the fastest propagator on 71% of instances. TBE can be up to 12 (98) times faster than the CT (CD) algorithms. In addition, the TBE propagator can respectively solve 270 and 23 more instances than the CD and CT propagators in 10 minutes.

We found that the more the reduction rules reduce the TBE (larger ESR), the greater the speedup of TBE over CD. The solver performance is shown in Figure 9.2b, we can see the solving time ratio between TBE and CD is consistent with ESR. On most instances with "100 $\leq$ESR", the TBE can be 50X faster than CD. We turn to the CT comparison, shown in Figure 9.2c, TBE is faster than CT on most of the instances and only slightly slower on a few. The average CPU time of Algorithm 8.1 is only 0.43s, thus, TBE can overall outperform the CT algorithm for non-trivial instances. Comparing with existing results, the results in [VLS18, VLS19] show that CD is outperformed by CT for table benchmarks. In contrast, our results here outperform both CD and CT on table benchmarks.

9.6 Related work

Cheng and Yap proposed the first MDD GAC algorithm MDDc [CY10] using a depth-first search strategy. Later, various incremental algorithms, such as incremental MDD [GSS11] and MDD4R [PR14], were proposed to avoid redundant traversal. Comparing to table GAC algorithms, MDD GAC algorithms can be more efficient when the MDD constraints are compact, but there is a large gap between the MDD representation's compression ratio and when it outperforms table propagators. Recent works show table GAC algorithms using bitwise operations can overall outperform MDD GAC algorithms on a large set of benchmarks [WXYL16, DHL$^+$16].

In order to reduce the gap, Verhaeghe et al. proposed the MDD GAC algorithm CD [VLS18] using ideas from CT. The CD algorithm has been also extended with some alternative representations of MDDs, such as semi-MDDs [VLS18] and basic smart MVDs [VLS19]. Although CD and its variants can be faster than the MDD4R algorithm and reduce the gap, CT still overall outperforms CD and its variants on a large set of benchmarks (see the experimental results in [VLS18, VLS19]). The BCT AC propagator proposed in this chapter is the first algorithm, enforcing GAC on MDD constraints, which is shown to overall outperform the state-of-the-art table GAC algorithm CT.

9.7 Conclusion

Ordered Multi-valued Decision Diagram (MDD) is more compact than its table representation, but the table GAC propagator CT still overall outperforms the state-of-the-art MDD GAC propagators on a large set of benchmarks. In order to reduce the large gap between the MDD representation's compression ratio and efficiency of MDD GAC propagators, we propose to encode MDD constraints as BCT constraints, where BCTs can be exponentially smaller than MDDs. We give a BCT AC propagator using bitwise operations to enforce GAC on the MDD constraints encoded as BCT constraints. Our experimental results on a large set of benchmarks show that BCT constraints are very compact and BCT AC propagator outperforms the state-of-the-art MDD GAC propagator CD and table GAC propagator CT.

Chapter 10

CNF Encodings of BCT Constraints

10.1 Introduction

In Constraint Programming (CP) solvers, Ordered Multi-valued Decision Diagram (MDD) constraints can be directly handled with MDD Generalized Arc Consistency (GAC) propagators, e.g., the MDDc [CY10] and CD [VLS18] algorithms. Alternatively, MDD constraints can also be solved with SAT solvers by encoding MDD constraints into CNF forms [AGMES16]. In this way, SAT solvers can directly solve the constraints which can be modelled as MDDs constraints[ANO+12, AS14, BCSV17]. SAT solvers have been shown to work very well on solving CSP instances. For example, the PicatSat solver which uses CNF encodings and SAT solvers won the first place in the XCSP competition 2019, 2022[1] and second place in the Minizinc challenge 2020, 2021, 2022.[2] In previous chapters, we have shown that BCT can be much more compact than MDD, so we show in this chapter that the CNF encodings of MDD constraints can be reduced and improved by use of BCT CNF encodings.

Binary constraints are similar to MDD constraints, which can also be encoded into CNF forms with a number of different CNF encodings, such as the support encoding [Kas90, Gen02], log encoding [IM94, Wal00, VG08] and direct encoding [Wal00]. In this chapter, we tailor the CNF encodings of binary constraints to encode Binary Constraint Trees (BCTs) into CNF forms, allowing SAT solvers to be used for BCT constraints, where BCT constraints can be more compact than MDD constraints.

We investigate five CNF encodings of BCT constraints, including the log encoding, direct encoding, support encoding and two new encodings: partial support encoding and minimal support encoding. We tailor three well-known CNF encodings of binary constraints, i.e., the log, direct and support encodings, to handle BCT constraints. In addition, we introduce the partial support encoding and minimal support encoding by eliminating clauses and Boolean variables from the support encoding of BCT constraints. Then we analyze the strength of unit propagation on these 5 CNF encodings of BCT constraints. The support

[1] http://www.xcsp.org/competitions/

[2] https://www.minizinc.org/challenge.html

encoding of BCT constraints, which implements propagation completeness [BMS12], can have a greater propagation strength than the other CNF encodings. The partial support encoding and minimal support encoding are more compact than the support encoding but their propagation strength is weaker than the support encoding. The log encoding and direct encoding, which do not implement weak consistency, have the weakest propagation strength. We also compare the five CNF encodings of BCT constraints and seven existing MDD encodings [AGMES16] using the Kissat (a state-of-art) SAT solver [FH20] on a range of existing benchmarks. Our experimental results show that the CNF encodings of BCT constraints can outperform MDD CNF encodings.

10.2 CNF encoding and unit propagation strength

A *CNF encoding* of a CSP $P = (X, C)$ is a CNF such that the CNF is satisfiable iff P is satisfiable, where a *CNF* is a CSP consisting of clauses and Boolean variables. Typically, a CNF encoding consists of a variable encoding (VE) and a constraint encoding (CE) where VE encodes the variables in X as a set φ^X of Boolean variables and each constraint c in C corresponds to a constraint (Boolean function) φ^c over the Boolean variables, while CE encodes φ^c as a CNF F^c over Boolean variables Y such that $scp(\varphi^c) \subseteq Y$ and $sol(F^c)[scp(\varphi^c)] = rel(\varphi^c)$. There have been many VEs which can be used to encode finite domain variables as Boolean variables, such as direct encoding and log encoding [Wal00]. In addition, there are also various CEs including several existing CNF encodings of MDD constraint [AGMES16]. A way to analyze a CE is evaluating the strength of unit propagation on the encoded constraint F^c. We will use the following 4 levels to classify the strength of unit propagation:

- F^c implements *weak consistency* if for any tuple τ over $scp(\varphi^c)$ such that $F^c|_\tau$ is unsatisfiable, some variable domains in $UP(F^c|_\tau)$ are empty.
- F^c implements *domain consistency* if for any tuple τ over a subset of $scp(\varphi^c)$ and a literal (v, a) in $UP(F^c|_\tau)$ such that $v \in scp(\varphi^c)$ and all variable domains in $UP(F^c|_\tau)$ are not empty, there is at least a solution of $F^c|_\tau$ including (v, a).
- F^c implements *unit refutation completeness* [DV94, AGMES16, KS19, KS20] if for any tuple τ over a subset of Y such that $F^c|_\tau$ is unsatisfiable, some variable domains in $UP(F^c|_\tau)$ are empty.
- F^c implements *propagation completeness* [BMS12, AGMES16, KS19, KS20] if for any tuple τ over a subset of Y and a literal l in $UP(F^c|_\tau)$ such that all variable domains in $UP(F^c|_\tau)$ are not empty, there is at least a solution of $F^c|_\tau$ including l.

where $F^c|_\tau$ is a CNF generated by assigning the tuple τ, and $UP(F^c|_\tau)$ is a CNF generated by enforcing GAC on $F^c|_\tau$ with the unit propagation (more details can be found in Sections

2.1 and 2.3.5). In this classification, propagation completeness is the strongest level, unit refutation completeness is incomparable with domain consistency, and weak consistency is the weakest level. Note that the definition of weak/domain consistency is defined for evaluating the strength of unit propagation and will be used in the rest of this chapter.

Example 10.1. *Assume the CNF* F^c *is* $(\{x, y, z\}, \{x \vee y, \neg y \vee z, x \vee \neg z\})$ *and the scope* $scp(\varphi^c)$ *is* $\{x, y\}$. $(x, false)$ *is the only literal in* F^c *which cannot be extended to a solution of* F^c. $(x, false)$ *is in* $UP(F^c)$, *so* F^c *does not implement propagation completeness. Then* x *is in* $scp(\varphi^c)$, *thus,* F^c *also does not implement domain consistency. For any tuple* τ *over a subset of* $\{x, y, z\}$, *if* $F^c|_\tau$ *is unsatisfiable,* $UP(F^c|_\tau)$ *has empty variable domains. So* F^c *implements unit refutation completeness and weak consistency.*

10.3 CNF encodings for binary constraints

In this section, we introduce three well-known CNF encodings, i.e., the log encoding [IM94, Wal00, VG08], direct encoding [Wal00] and support encoding [Kas90, Gen02], which are used to transform any binary CSP (X, C) into CNF. These CNF encodings represent each variable x in X as a set of Boolean variables such that every value in $\mathcal{D}(x)$ corresponds to exactly one tuple over the variables. In addition, for each binary constraint $c \in C$, the encodings use clauses to represent the tuples τ over $scp(c)$ such that $\tau \notin rel(c)$ (or $\tau \in rel(c)$). These CNF encodings can be directly used to encode BCT constraints, since any BCT is also a binary CSP.

10.3.1 Log encoding

Every variable $x \in X$ is represented as $k = \lceil \log_2(d) \rceil$ Boolean variables $V^x = \{v_1^x, ..., v_k^x\}$, where $d = |\mathcal{D}(x)|$. Let T be the set of the first d assignments over B^x in the lexicographic order. Every value a in $\mathcal{D}(x)$ corresponds to a tuple τ_a in T. In addition, if $|\mathcal{D}(x)|$ is not a power of two, the assignments over V^x which are not in T can be excluded by adding the following clause [VG08] for each literal $(v_i^x, false)$ in the d^{th} (last) tuple τ in T

$$f(v_1^x) \vee \cdots \vee f(v_{i-1}^x) \vee \neg v_i^x \qquad where \quad f(v_j^x) = \begin{cases} v_j^x & \text{if } (v_j^x, false) \in \tau \\ \neg v_j^x & \text{if } (v_j^x, true) \in \tau \end{cases}$$

For each constraint $c \in C$ and an assignment $\{(x, a), (y, b)\}$ over $scp(c)$, if the tuple is not in $rel(c)$, then the following clause is added to exclude the tuple

$$\Big(\bigvee_{(v_j^x, true) \in \tau_a \cup \tau_b} \neg v_j^x \Big) \vee \Big(\bigvee_{(v_j^x, false) \in \tau_a \cup \tau_b} v_j^x \Big)$$

Example 10.2. *The log encoding of the binary CSP* $P = (\{x_1, x_2, x_3, x_4\}, \{x_1 + x_3 \leq 5, x_3 = 3 \vee x_4 = 3, x_2 + x_4 \leq 5\})$*, where variable domains are* $\{1, 2, 3\}$*, consists of 8 Boolean variables* $\{v_1^{x_1}, v_2^{x_1}, v_1^{x_2}, v_2^{x_2}, v_1^{x_3}, v_2^{x_3}, v_1^{x_4}, v_2^{x_4}\}$ *and 10 clauses:*

$$\neg v_1^{x_1} \vee \neg v_2^{x_1} \qquad \neg v_1^{x_2} \vee \neg v_2^{x_2} \qquad \neg v_1^{x_3} \vee \neg v_2^{x_3} \qquad \neg v_1^{x_4} \vee \neg v_2^{x_4}$$
$$\neg v_1^{x_1} \vee v_2^{x_1} \vee \neg v_1^{x_3} \vee v_2^{x_3} \qquad \neg v_1^{x_2} \vee v_2^{x_2} \vee \neg v_1^{x_4} \vee v_2^{x_4} \qquad v_1^{x_3} \vee v_2^{x_3} \vee v_1^{x_4} \vee v_2^{x_4}$$
$$v_1^{x_3} \vee \neg v_2^{x_3} \vee v_1^{x_4} \vee v_2^{x_4} \qquad v_1^{x_3} \vee v_2^{x_3} \vee v_1^{x_4} \vee \neg v_2^{x_4} \qquad v_1^{x_3} \vee \neg v_2^{x_3} \vee v_1^{x_4} \vee \neg v_2^{x_4}$$

10.3.2 Direct encoding

Every variable $x \in X$ is represented as d Boolean variable $B^x = \{v_{a_i}^x | a_i \in \mathcal{D}(x)\}$, where $\mathcal{D}(x) = \{a_1, \cdots, a_d\}$. In addition, an exactly-one constraint (EO) over B^x is introduced to guarantee that if a variable in B^x is assigned with *true*, then the other variables in B^x are assigned with $false$. The exactly-one constraint is encoded with *ladder encoding* [GN04] which includes $d-1$ additional Boolean variables $A^x = \{w_1^x, \cdots, w_{d-1}^x\}$ and a set of $EO(x)$ clauses:

$$\neg v_{a_1}^x \vee \neg w_1^x \qquad v_{a_1}^x \vee w_1^x \qquad v_{a_d}^x \vee \neg w_{d-1}^x \qquad \neg v_{a_d}^x \vee w_{d-1}^x$$
$$\{w_{i-1}^x \vee \neg w_i^x, v_{a_i}^x \vee w_i^x \vee \neg w_{i-1}^x, \neg v_{a_i}^x \vee \neg w_i^x, \neg v_{a_i}^x \vee w_{i-1}^x | 2 \leq i \leq d-1\}$$

The latter encoding implements propagation completeness [BMS12, KS19]. The clauses in $EO(x)$ can guarantee that every value a_i in $\mathcal{D}(x)$ corresponds to exactly one solution $t(x, a_i)$ of the CNF $(B^x \cup A^x, EO(x))$, where $t(x, a_i) = \{(v_b^x, false) | b \in \mathcal{D}(x), b \neq a_i\} \cup \{(v_{a_i}^x, true)\} \cup \{(w_j^x, true) | 1 \leq j < i\} \cup \{(w_j^x, false) | i \leq j < d\}$. Then for each $c \in C$ and an assignment $\tau = \{(x, a_i), (y, b_j)\}$ over $scp(c)$ such that $\tau \notin rel(c)$, the clause $\neg v_{a_i}^x \vee \neg v_{b_j}^y$ is added.

Example 10.3. *The direct encoding of the binary CSP given in Example 10.2 includes 20 Boolean variables and the following clauses:*

$$\{\neg v_1^{x_j} \vee \neg w_1^{x_j}, v_1^{x_j} \vee w_1^{x_j}, v_3^{x_j} \vee \neg w_2^{x_j}, \neg v_3^{x_j} \vee w_2^{x_j} | 1 \leq j \leq 4\}$$
$$\{w_1^{x_j} \vee \neg w_2^{x_j}, v_2^{x_j} \vee w_2^{x_j} \vee \neg w_1^{x_j}, \neg v_2^{x_j} \vee \neg w_2^{x_j}, \neg v_2^{x_j} \vee w_1^{x_j} | 1 \leq j \leq 4\}$$
$$\neg v_3^{x_1} \vee \neg v_3^{x_3} \quad \neg v_3^{x_2} \vee \neg v_3^{x_4} \quad \neg v_1^{x_3} \vee \neg v_2^{x_4} \quad \neg v_1^{x_3} \vee \neg v_1^{x_4} \quad \neg v_2^{x_3} \vee \neg v_1^{x_4} \quad \neg v_2^{x_4} \vee \neg v_2^{x_4}$$

10.3.3 Support encoding

Every variable $x \in X$ is represented as d Boolean variable $B^x = \{v^x_{a_i} | a_i \in \mathcal{D}(x)\}$, where $\mathcal{D}(x) = \{a_1, \cdots, a_d\}$, and an exactly-one constraint over B^x is used to make sure that each value in $\mathcal{D}(x)$ corresponds to exactly one tuple over B^x. The exactly-one constraint is encoded with $(A^x \cup B^x, EO(x))$, i.e., ladder encoding. In addition, for each value $a \in \mathcal{D}(x)$ and a constraint $c \in C$ such that $scp(c) = \{x, y\}$, a clause $cl(x, a, c)$ is added where

$$cl(x, a, c) = \neg v^x_a \vee (\bigvee_{\{(x,a),(y,b)\} \in rel(c) \wedge b \in \mathcal{D}(y)} v^y_b)$$

By using the clauses $cl(x, c) = \{cl(x, a, c) | c \in C, x \in scp(c), a \in \mathcal{D}(x)\}$, unit propagation on the support encoding of a binary CSP can achieve AC on the binary CSP [Gen02].

10.4 CNF encoding for BCT constraints

For any BCT constraint (V, P), the CNF encodings of binary constraints can be directly used to encode the BCT constraint, because the BCT P is a binary CSP. In addition, binary constraints are special cases of BCT constraints, i.e., every binary constraint is a BCT constraint without any hidden variables. When comparing the unit propagation, there is a subtlety, the strength of unit propagation on the CNF encodings of BCT constraints can be different from that for binary constraints. In this section, we will discuss the strength of unit propagation when using the log, direct and support encodings to encode BCT constraints as CNF. Afterwards, we will propose two further CNF encodings of BCT constraints by eliminating Boolean variables and clauses from the support encoding of the constraints.

10.4.1 Encodings from binary constraints

The log encoding of binary constraints implements weak consistency but we highlight that it does not do so for BCT constraints, since BCT constraints can include hidden variables. Assume P is the binary CSP given in Example 10.2 and F is the log encoding of the BCT constraint $(\{x_1, x_2\}, P)$. Then F does not implement weak consistency. For example, the tuple τ=$\{(v^{x_1}_1, true), (v^{x_1}_2, false), (v^{x_2}_1, true), (v^{x_2}_2, false)\}$ is not included by any solution of F, i.e., $F|_\tau$ is unsatisfiable but $UP(F|_\tau)$ does not include any empty variable domain. Therefore, the log encoding of BCT constraints does not implement weak consistency.

Proposition 10.1. *Log encoding of BCT constraints does not implement weak consistency.*

Similarly, the direct encoding of BCT constraints also does not implement weak consistency. For example, the tuple τ=$\{(v^{x_1}_3, true), (v^{x_2}_3, true)\}$ is not included by any solution of

the direct encoding F (given in Example 10.3) of the BCT constraint $(\{x_1, x_2\}, P)$, i.e., $F|_\tau$ is unsatisfiable, but $UP(F|_\tau)$ does not include any empty variable domain.

Proposition 10.2. *Direct encoding of BCT constraints does not implement weak consistency.*

Unit propagation on the support encoding of a binary CSP can achieve AC on the binary CSP. Further, the constraint graph of a BCT is a tree. Hence, unit propagation on the support encoding of a BCT constraint can achieve AC on the BCT constraint. For any BCT (X, C), we can set a variable $x \in X$ as the root and construct a tree order O over X such that $O_1 = x$, where O is a *tree order* if for any $j > 1$ and $O_j \in X$, there is exactly one constraint $c \in C$ such that $scp(c) = \{O_i, O_j\}$ and $i < j$. In addition, we use T^O to denote a set of clauses $\bigcup\{cl(O_i, c)|c \in C, scp(c) = \{O_i, O_j\}, i < j\}$ with respect to a tree order O.

Lemma 10.1. *Given a BCT $P = (X, C)$ and a tree order O over X, if a literal $(v_a^{O_1}, true)$ is included in $UP(F)$ and all variable domains in $UP(F)$ are not empty, there is $\tau \in sol(P)$ such that $(O_1, a) \in \tau$ and $(v_b^x, true)$ is included in F for all $(x, b) \in \tau$, where $F = (A, C^A)|_{\tau'}$ and $B^x \subseteq A$ for all $x \in X$ and $T^O \subseteq C^A$ and τ' is a tuple over a subset of A.*

Proof. For any $c \in C$ where $scp(c) = \{O_i, O_j\}$ and $i < j$, if a literal $(v_a^{O_i}, true)$ is included in $UP(F)$, there must be a literal $(v_b^{O_j}, true)$ in $UP(F)$ such that $\{(O_i, a), (O_j, b)\} \in rel(c)$, otherwise unit propagation with the clause $cl(O_i, a, c)$ can remove $(v_a^{O_i}, true)$ from $UP(F)$. Note that the clause $cl(O_i, a, c)$ encodes the implication: if $v_b^{O_j} = false$ for all tuple $\{(O_i, a), (O_j, b)\} \in rel(c)$, then $v_a^{O_i} = false$.

So we can construct a series of tuples $\{\tau_1, ..., \tau_n\}$ such that $n = |X|$ and $\tau_1 = \{(O_1, b_1)\}$ and $b_1 = a$ and for $j > 1$, $\tau_j = \tau_{j-1} \cup \{(O_j, b_j)\}$ and $(v_{b_j}^{O_j}, True)$ is included in $UP(F)$ and $\{(b_i, O_i), (b_j, O_j)\} \in rel(c)$, where c is the only constraint in C such that $scp(c) = \{O_i, O_j\}$ and $i < j$. The tuple τ_n is a solution of P and $(O_1, a) \in \tau_n$. □

We now show the support encoding of BCT constraints implements propagation completeness.

Proposition 10.3. *The support encoding $F = (A \cup B, T \cup E)$ of BCT constraints (V, P) implements propagation completeness, where P is a BCT (X, C) and $A = \bigcup_{x \in X} A^x$ and $B = \bigcup_{x \in X} B^x$ and $T = \{cl(x, c)|c \in C, x \in scp(c)\}$ and $E = \bigcup_{x \in X} EO(x)$.*

Proof. Assume $F^\tau = UP(F|_\tau)$ and all variable domains in F^τ are not empty where τ is a tuple over a subset of $A \cup B$. The ladder encoding implements propagation completeness, therefore, for any $x \in X$, if F^τ includes a literal l of a variable in $A^x \cup B^x$, then there is a tuple $t(x, a)$ such that F^τ includes $t(x, a)$ and $t(x, a) \in sol(A^x \cup B^x, EO(x))$ and $l \in t(x, a)$ and $(v_a^x, true) \in t(x, a)$. We can set x as root and construct a tree order O over X such that $O_1 = x$, thus, there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including l (based on Lemma 10.1). So F implements propagation completeness. □

10.4.2 Partial support encoding

We now introduce a new CNF encoding of BCT constraints, called partial support encoding, by eliminating clauses from the support encoding of the BCT constraints. Give any BCT $P = (X, C)$ and variables $V \subseteq X$, the *partial support encoding* of the BCT constraint (V, P) is a CNF $F = (A^V \cup B^V \cup B^H, T \cup E^V)$, where $A^V = \bigcup_{x \in V} A^x$ and $B^V = \bigcup_{x \in V} B^x$ and $B^H = \bigcup_{h \in X \setminus V} B^h$ and $T = \{cl(x, c, a) | c \in C, x \in scp(c), a \in \mathcal{D}(x)\}$ and $E^V = \bigcup_{x \in V} EO(x)$. Partial support encoding has the same Boolean variables and clauses as the support encoding of (V, P), except that the clauses in $EO(h)$ and the Boolean variables in A^h are removed from the support encoding of (V, P) for any hidden variables h in $X \setminus V$.

For each solution τ of P, we can construct a solution of F, e.g., $(\bigcup_{(x,a) \in \tau} t(x, a)) \cup \{(v_a^x, true) | (x, a) \in \tau\} \cup \{(v_a^x, false) | (x, a) \notin \tau, a \in \mathcal{D}(x)\}$. Conversely, every solution τ of F corresponds to at least one solution of P (see Lemma 10.1), since variable domains in $UP(F|_\tau)$ are not empty. However, the strength of unit propagation on the partial support encoding is weaker than that on the support encoding for BCT constraints. Partial supporting encoding implements domain consistency and unit refutation completeness but not propagation completeness.

For the partial support encoding F of a BCT $(\{x_1, x_2\}, P)$ where P is the binary CSP (BCT) given in Example 10.2, the literal $(v_3^{x_3}, false)$ cannot be extended to any solution of the CNF $F|_\tau$ but $(v_3^{x_3}, false)$ is in $UP(F|_\tau)$, where $\tau = \{(v_3^{x_4}, false), (v_1^{x_3}, false), (v_2^{x_3}, false), (v_3^{x_1}, false)\}$ is a tuple over the Boolean variables $\tau = \{v_3^{x_4}, v_1^{x_3}, v_2^{x_3}, v_3^{x_1}\}$. Therefore, partial support encoding of BCT constraint does not implement propagation completeness.

Proposition 10.4. *Partial support encoding for BCT constraints does not implement propagation completeness.*

Partial support encoding implements unit refutation completeness (based on Lemma 10.1), thus, it also implements weak consistency.

Proposition 10.5. *The partial support encoding $F = (A^V \cup B^V \cup B^H, T \cup E^V)$ of a BCT constraint (V, P) implements unit refutation completeness where $P = (X, C)$.*

Proof. Let $x \in X$ and $F^\tau = UP(F|_\tau)$ where τ is a tuple over a subset of $A^V \cup B^V \cup B^H$. If all variable domains in F^τ are not empty, there is $a \in \mathcal{D}(x)$ such that $(v_a^x, true)$ is in F^τ, otherwise unit propagation with $EO(x)$ can remove all values of the variables in $A^x \cup B^x$, since the ladder encoding implements propagation completeness and every tuple in $sol(A^x \cup B^x, EO(x))$ includes at least a value *true* of a variable in B^x. Therefore, there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including $(v_a^x, true)$ by setting x as root (based on Lemma 10.1). So F implements unit refutation completeness. □

In addition, from Lemma 10.1, we can also get that partial support encoding implements domain consistency.

Proposition 10.6. *The partial support encoding* $F = (A^V \cup B^V \cup B^H, T \cup E^V)$ *of a BCT constraint* (V, P) *implements domain consistency where* $P = (X, C)$.

Proof. Let $x \in V$ and $F^\tau = UP(F|_\tau)$ where τ is a tuple over a subset of $A^V \cup B^V$ and all variable domains in F^τ are not empty. If a literal l of a variable in $A^x \cup B^x$ is included in F^τ, there is a value $a \in \mathcal{D}(x)$ such that $l \in t(x, a)$ and $(v_a^x, true) \in t(x, a)$ and $(v_a^x, true)$ is included in F^τ (since ladder encoding implements propagation completeness). So there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including l (Lemma 10.1 by setting x as root). Hence, the partial support encoding implements domain consistency. □

10.4.3 Minimal support encoding

We now give a more compact CNF encoding of BCT constraints called minimal support encoding. Give any BCT $P = (X, C)$ and variables $V \subseteq X$, the *minimal support encoding* of the BCT constraint (V, P) with respect to a tree order O over X is a CNF $F = (A^V \cup B^V \cup B^H, T^O \cup E^V)$, where $P = (X, C)$ and $A^V = \bigcup_{x \in V} A^x$ and $B^V = \bigcup_{x \in V} B^x$ and $B^H = \bigcup_{h \in X \setminus V} B^h$ and $E^V = \bigcup_{x \in V} EO(x)$ and $O_1 \in V$. Minimal support encoding has the same Boolean variables as the partial support encoding but the minimal support encoding does not include the clauses $cl(O_j, c)$ for any binary constraint $c \in C$ between 2 variables $O_i, O_j \in X$ such that $i < j$.

The strength of unit propagation on minimal support encoding is weaker than that on partial support encoding. For the minimal support encoding F of a BCT $(\{x_1, x_2\}, P)$ with respect to a tree order $x_1 < x_3 < x_4 < x_2$ and a tuple $\tau = \{(v_3^{x_1}, true)\}$ where P is the binary CSP given in Example 10.2, the literal $(v_3^{x_4}, true)$ cannot be extended to a solution of the CNF $F|_\tau$ but $(v_3^{x_4}, true)$ is included in $UP(F|_\tau)$. So the minimal support encoding does not implement domain consistency and propagation completeness.

Proposition 10.7. *Minimal support encoding for BCT constraints does not implement domain consistency and propagation completeness.*

In addition, the minimal support encoding is stronger than the log and direct encodings for BCT constraints. From Lemma 10.1, we can get that the minimal support encoding implements unit refutation completeness and weak consistency.

Proposition 10.8. *The minimal support encoding* F *of a BCT constraint* (V, P) *with respect to a tree order* O *implements unit refutation completeness where* $x = O_1$ *and* $P = (X, C)$.

Proof. Let $F^\tau = UP(F|_\tau)$ where τ is a tuple over a subset of $A^V \cup B^V \cup B^H$. If all variable domains in F^τ are not empty, there is $a \in \mathcal{D}(x)$ such that $(v_a^x, true)$ is in F^τ, otherwise unit propagation with $EO(x)$ can remove all values of the variables in $A^x \cup B^x$, since ladder encoding implements propagation completeness and every tuple in $sol(A^x \cup B^x, EO(x))$

includes at least a value *true*. Therefore, there is a solution of P including (x, a) which corresponds to a solution of $F|_\tau$ including $(v_a^x, true)$ based on Lemma 10.1 (where x is set as root). So F implements unit refutation completeness. □

	Log	Direct	Minimal Support	Partial Support	Support
Weak consistency	✗	✗	✓	✓	✓
Domain consistency	✗	✗	✗	✓	✓
Unit refutation completeness	✗	✗	✓	✓	✓
Propagation completeness	✗	✗	✗	✗	✓

Table 10.1: Strength of Unit Propagation on various encodings of BCT constraints. The label ✓(✗) denotes the CNF encodings implement (does not implement) a strength level.

Table 10.1 summarizes the strength of unit propagation on all five CNF encoding of BCT constraints. The support encoding of BCT constraints, which implements propagation completeness, is the strongest encoding. Then the partial support encoding implementing domain consistency is stronger than the log, direct and minimal support encodings. In addition, the log and direct encodings of BCT constraints are weaker than the minimal support encoding, since the log and direct encodings of BCT constraints do not implement weak consistency.

10.5 Experiments

The BCT encodings differ in the number of variables and clauses, not only in their propagation strength, and we will see in the experiments that they behave differently on different instances. We evaluate our five CNF encodings of BCT constraints, i.e., the log encoding, direct encoding, support encoding, PS (partial support) encoding and MS (minimal support) encoding, with seven existing CNF encodings [AGMES16] of MDD constraints, i.e., the Min (minimal), GMin (GenMiniSAT), Tes (Tseitin), BaP (Basic path), LevP (level path), NNFP (NNF path) and ComP (complete path) encodings. We employ the Kissat SAT solver [FH20] in default configuration to solve the resulting CNF. We also test the BCT AC propagator given in Chapter 9 with the Abscon solver [MLB01], where the Abscon solver uses the binary branching MAC and geometric restart strategy[3], lexical value heuristic and five choices of variable heuristics (Lexical, DDeg [SG97], WDeg [BHLS04], Activity [MVH12] and Impact [Ref04]). We highlight the Abscon results are to compare CNF encodings with a SAT solver with a BCT AC propagator in a CP solver.

[3]The initial $cutoff = 10$ and $\rho = 1.1$. For each restart, $cutoff$ is the allowed number of failed assignments and $cutoff$ increases by $(cutoff \times \rho)$ after restart.

Acronym	CNF encoding	Constraint type
PS	The partial support encoding	BCT constraints
MS	The minimal support encoding	
Min	The minimal encoding	MDD constraints
GMin	The GenMiniSAT encoding	
Tes	The Tseitin encoding	
BaP	The basic path encoding	
LevP	The level path encoding	
NNFP	The NNF path encoding	
ComP	The complete path encoding	

Table 10.2: Acronyms of various CNF encodings.

Experiments were run on a 3.20GHz Intel i7-8700 machine. Solving time is limited to 10 minutes per instance and memory to 12G. We tested the encodings with existing benchmarks also used by other papers [GSS11, CXY12], and instances from the 2019 XCSP competition[4] and Minizinc challenge 2021.[5]

Tables 10.3, 10.4, 10.5 and 10.6 show the average solving times (in seconds) of the CNF encodings and the Abscon solver, and if there are some instances which cannot be solved in 10 minutes, then the tables give the number of unsolvable instances, i.e., the number of time-out or memory-out instances. The "Inst." and "# I" columns respectively give the names of different instance series and the numbers of CSP instances used. The "# Sol" row shows the total number of instances solved in 10 minutes. The best results of all methods are in bold, and the underlined results are the best results of the CNF encodings. The "Itime" row is the average initialization time (in seconds) of different methods and includes the encoding times. In addition, the "Itime" row also shows the number of memory-out instances if the CNF encoding runs out of memory during initialization. For different benchmarks, the Abscon solver results use the best variable heuristic from the 5 heuristics. The acronyms of various CNF encodings are given in Table 10.2.

Figures 10.1a, 10.2a, 10.3a and 10.4a compare the performance profiles [DM02] of the methods VB-BCT, VB-MDD and VB-Abscon, where VB-BCT and VB-MDD respectively denote the virtual best CNF encoding of BCT constraints and MDD constraints respectively. VB-Abscon is the Abscon solver using the virtual best variable heuristic. The y-axis is the

[4] http://www.cril.univ-artois.fr/XCSP19/

[5] https://www.minizinc.org/challenge.html

percentage of instances solved and the x-axis is the ratio of the solving time of a method to the virtual best method (time ratio). In the figures, we remove (i) the trivial instances which can be solved by all methods (VB-BCT, VB-MDD, and VB-Abscon) in 2 seconds and (ii) the instances which cannot be solved by any method in 10 minutes. Figures 10.1b, 10.2b, 10.3b and 10.4b use scatter plots to compare the solving times of various CNF encodings. In the figures, each dot denotes a CSP instance of a series, and the dot shapes correspond to instance series. In order to use logarithmic scales, the time on the x/y-axis is set as (1 + solving time). Figures 10.1c, 10.2c, 10.3c and 10.4c compare the number of clauses of different CNF encodings given by the x and y-axis.

10.5.1 Benchmark Series 1: NFA

We use the 6 NFA series which are also used in [CXY12]. These NFA benchmarks are modelled with NFA constraints. The NFA constraints can be transformed into BCT constraints and also MDD constraints. The direct tree binary encoding introduced in Chapter 7 is used to encode NFA constraints as BCT constraints, and then the BCT constraints are further reduced with the reduction rules proposed in Chapter 8. In addition, the automaton library `dk.brics.automaton`[6] is used to minimize and transform any NFA into a DFA, where the DFA is directly expanded into the corresponding (quasi-reduced [AGMES16]) MDD.

		BCT					MDD							Abscon
Inst.	#I	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(DDeg)
NFA-50	14	1 out	166.42	11.62	7.76	4.86	7.91	2 out	3 out	102.18	126.79	109.96	107.57	**1.21**
NFA-36	18	16 out	10 out	61.69	30.40	19.53	1 out	15 out	15 out	11 out	11 out	11 out	13 out	**4.98**
NFA-34	13	89.63	14.57	2.20	1.38	1.25	1.78	16.56	24.90	19.59	25.25	19.16	21.17	**0.54**
NFA-54	15	15 out	14 out	179.23	91.95	69.11	2 out	15 out	15 out	15 out	15 out	15 out	15 out	**11.62**
NFA-57	15	14 out	10 out	1 out	55.42	27.07	1 out	13 out	15 out	10 out	11 out	11 out	12 out	**6.40**
NFA-60	15	14 out	6 out	30.13	16.80	10.13	25.79	12 out	15 out	2 out	3 out	3 out	2 out	**2.50**
#Sol	90	30	50	89	**90**	**90**	86	33	27	52	50	50	48	**90**
Itime	90	0.51	0.38	0.25	0.23	0.23	290.65	292.62	297.76	2 out	6 out	12 out	13 out	2.10

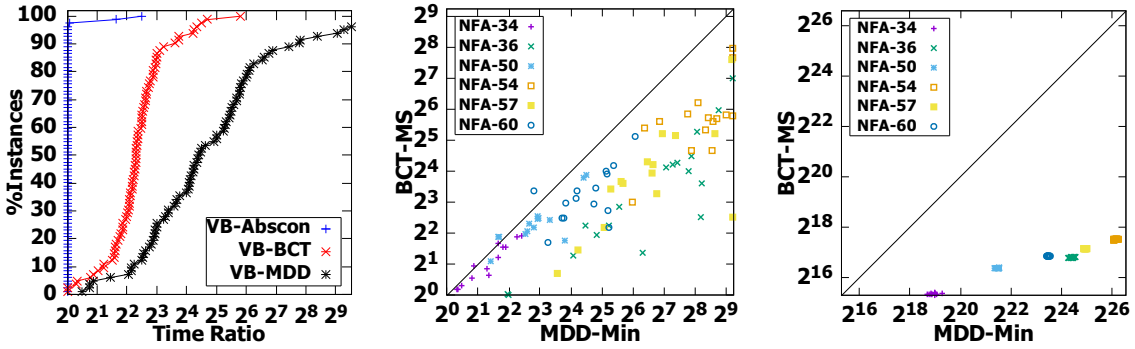
Table 10.3: NFA benchmarks.

Table 10.3 shows the average result of the NFA instances. The NFA constraints used in the instances are much smaller than the corresponding MDDs, and the resulting encodings of BCT constraints also fit in memory. However, for MDDs being larger, the MDD CNF encodings BaP, LevP, NNFP and ComP run out of memory (memory-out).[7] For these

[6] http://www.brics.dk/automaton/

[7] The ComP encoding of an instance including 156284122 clauses and 42599669 variables can still fit in

encodings, there are 2, 6, 12 and 13 memory-out NFA instances in the NFA-54 series, respectively. We remark that if an CNF encoding becomes too large for an instance, it simply cannot be solved with a SAT solver. The average initialization time of encoding MDD constraints are much larger than that of encoding BCT constraints, e.g., the Itime of Min is 290 seconds but that of MS is less than 1 second. The CNF encodings of BCT constraints are much faster than those of MDD constraints. The MS and PS encodings can solve all 90 NFA instances in 10 minutes but the best CNF encoding of MDD constraints, i.e., the Min encoding, only solves 86 instances. The best result for this series is the Abscon propagator with the DDeg variable heuristic.



(a) Virtual best comparison **(b)** BCT-MS vs MDD-Min **(c)** The number of clauses

Figure 10.1: NFA benchmarks.

Figure 10.1a shows that the best overall result for the NFA instances is the VB-Abscon propagator followed by the CNF encodings where VB-BCT overall outperforms VB-MDD on solving the NFA instances. The MS encoding is more compact than the Min encoding, where Min has the best CNF encoding result of MDD constraints for these NFA benchmarks. For example, the number of clauses in MS can be up to 400 times less than that in Min (Figure 10.1c gives the overall comparison). Correspondingly, MS has potential to be faster than Min. Figure 10.1b shows that MS is faster than Min on almost all tested NFA instances.

10.5.2 Benchmark Series 2: Pentominoes

We use all 192 Pentominoes instances from the the pentominoes generator website.[8] Some of the Pentominoes instances were also used in the Minizinc challenge 2021. The instances are separated into 4 series, P-5, P-10, P-15 and P-20, where P-k denotes the instances using a $k \times k$ board. The constraints used in these benchmarks are represented with regular expressions (see [Lag08] for more details). We use the dk.brics.automaton library to encode

memory, thus, the memory-out CNF encodings are very large.

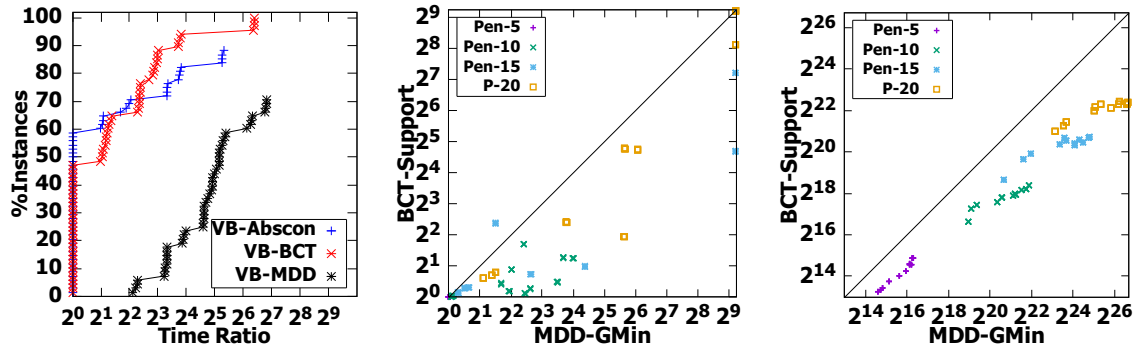
[8] https://github.com/zayenz/minizinc-pentominoes-generator

any regular expression into a DFA, and then directly expand the DFA into a MDD. The encoding from Chapters 7 and 8 is used to transform any MDD constraint into a BCT constraint.

		BCT					MDD							Abscon
Inst.	#I	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(Lex.)
P-5	48	0.27	**<0.01**	**<0.01**	**<0.01**	**<0.01**	0.05	0.01	0.06	0.02	0.02	0.02	0.02	0.02
p-10	48	154.70	32.04	**0.60**	0.76	3.62	72.86	4.94	31.65	32.08	17.73	19.42	18.28	9.95
P-15	48	36 out	24 out	**16 out**	20 out	20 out	24 out	24 out	24 out	24 out	24 out	24 out	28 out	20 out
P-20	48	44 out	24 out	16 out	16 out	16 out	24 out	20 out	32 out	36 out	36 out	24 out	28 out	**12 out**
#Sol	192	112	144	**160**	156	156	144	148	136	132	132	144	136	**160**
Itime	192	4.37	3.94	3.39	3.38	3.36	1.23	1.52	2.20	2.48	2.62	2.56	2.73	5.43

Table 10.4: Pentominoes benchmarks.

Table 10.4 gives the average result of the Pentominoes instances. The BCT support, PS and MS CNF encodings significantly outperform the MDD CNF encodings. The support encoding is faster or solves more instances than the other CNF encodings on all 4 series. For the MDD CNF encodings, GMin has the best overall result, but the support encoding of BCT constraints can solve 12 more instances than GMin. In addition, the CNF encodings of BCT constraints can be competitive with the Abscon solver. The support encoding gives the best performance on 3 out of 4 Pentominoes series.



(a) Virtual best comparison **(b)** BCT-Support vs MDD-GMin **(c)** The number of clauses

Figure 10.2: Pentominoes benchmarks.

Figure 10.2a shows the overall result on Pentominoes. VB-BCT is the best on more than 40% instances, and it can solve 5% more instances than VB-Abscon. Different CNF encodings of BCT constraints solve different instances, thus, VB-BCT can solve more instances than VB-Abscon. From Figure 10.2c, we can see that the number of clauses of GMin can be much more (up to 20 times more) than that of the support encoding. In addition, GMin is also

slower than the support encoding on almost all instances (see Figure 10.2b).

10.5.3 Benchmark Series 3: Nurse scheduling

We use four different models of the nurse scheduling problem, namely, N-1, N-2, N-3 and N-4. The nurse scheduling problems come from [BC94a, GSS11], where nurses are assigned with a day shift, evening shift, night shift or day off for each day. The models have a cardinality constraint [Rég96] per shift and a regular constraint per nurse. The cardinality constraints are used to guarantee that there are enough nurses to meet a demand of each shift. Each model has its own regular constraints as follows:

- In N-1, the model uses regular constraints to restrict that for each 7 days, a nurse work 1 or 2 night shifts, 1 or 2 evening shifts, 1 to 5 day shifts and 2 to 5 days off.
- In N-2, a nurse works 1 or 2 night shifts every 7 days, and 1 or 2 days off every 5 days.
- In N-3, a nurse works 1 or 2 night shifts every 9 days, and 2 or 3 days off every 7 days.
- In N-4, a nurse works 1 or 2 night shifts every 11 days, and 3 or 4 days off every 9 days.

All models restrict that a nurse can only work a second shift after 12 hours of the first. The cardinality and regular constraints are encoded as MDD constraints, and then the MDD constraints are transformed into BCT constraints. For each model, we use 50 instances from the N30 series[9] where the number of nurses of an instance is set to the maximum number of the nurse demand for a day.

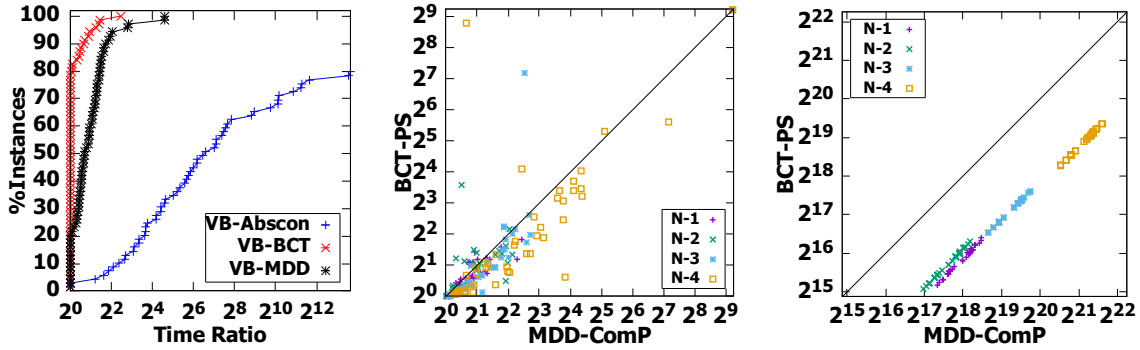
		BCT					MDD							Abscon
Inst.	#I	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(Act.)
N-1	50	1 out	12.88	**0.30**	0.49	3.10	2 out	0.50	0.84	1.09	1.29	0.77	0.64	7 out
N-2	50	11.34	1.86	**0.44**	0.80	0.84	1 out	0.48	0.63	2.95	0.83	0.71	0.72	14 out
N-3	50	2 out	2 out	**1 out**	**1 out**	2 out	4 out	2 out	3 out	**1 out**	**1 out**	**1 out**	**1 out**	13 out
N-4	50	50 out	10 out	3 out	**2 out**	5 out	5 out	4 out	5 out	4 out	**2 out**	3 out	**2 out**	15 out
#Sol	200	147	188	196	**197**	193	188	194	192	195	**197**	196	**197**	151
Itime	200	2.44	1.76	0.57	0.56	0.56	0.45	0.47	0.47	0.49	0.52	0.50	0.51	0.82

Table 10.5: Nurse scheduling benchmarks.

Table 10.5 shows that CNF encodings with the Kissat solver can overall outperform the Abscon solver for the nurse scheduling instances. The Abscon solver only solves 151/200 instances within timeout but the CNF encodings can solve 197/200 instances. We conjecture

[9] https://www.projectmanagement.ugent.be/nsp.php

that the performance benefit is because SAT solvers have clause learning but Abscon does not. In our detailed results, the CNF encodings of BCT constraints are faster than the CNF encodings of MDD constraints on most instances. For example, the PS encoding is faster than each CNF encoding of MDD constraints on most nurse scheduling instances.



(a) Virtual best comparison **(b)** BCT-PS vs MDD-ComP **(c)** The number of clauses

Figure 10.3: Nurse scheduling benchmarks.

From Figure 10.3a, we see that VB-BCT is the fastest method on more than 80% instances. VB-BCT can solve 20% more instances than VB-Abscon. The CNF encodings of BCT constraints have better performance than those of MDD constraints. For example, the number of clauses of the PS encoding is around 4 times less than that of the ComP encoding (shown in Figure 10.3c), and the PS encoding can be faster than ComP on more than 85% instances (see Figure 10.3b), where ComP is the best CNF encoding of MDD constraints for the nurse scheduling instances.

10.5.4 Benchmark Series 4: XCSP

We use five instance series from the XCSP website[10] as they are BDD/MDD instances: bdd-15, bdd-18, mdd-p5 (MDD-half), mdd-p7 (MDD-0.7) and mdd-p9 (MDD-0.9). Some of these instances were also used in the 2019 XCSP competition. The instances bdd-15 and bdd-18 are introduced in [CY06c], and then the instances mdd-p5, mdd-p7 and mdd-p9 are introduced in [CY10, XY13], where mdd-pk is a MDD with sharing probability $\frac{k}{10}$ (see [CY10, XY13] for more details).

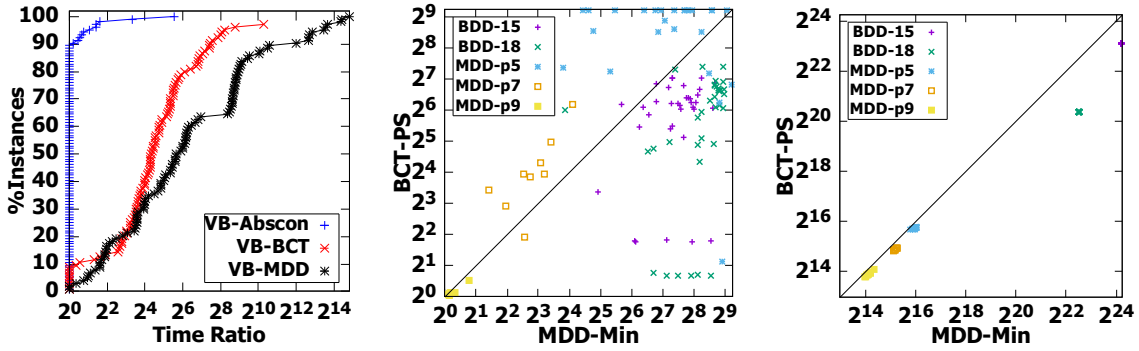
Table 10.6 shows that the Abscon solver using the Activity heuristic is the fastest overall for these instances. The Abscon solver can solve all instances while the CNF encodings are time-out on some instances. The CNF encodings of BCT and MDD constraints perform better on different instances. On the bdd-15, bdd-18 and mdd-p9 series, the PS encoding is faster than the CNF encodings of MDD constraints while Min is the best CNF encoding on

[10] http://xcsp.org

		BCT					MDD							Abscon
Inst.	#	Log	Direct	Support	PS	MS	Min	GMin	Tes	BaP	LevP	NNFP	ComP	(Act.)
bdd-15	35	35 out	222.99	106.44	67.67	33.05	192.29	152.37	16 out	22 out	29 out	28 out	33 out	**2.42**
bdd-18	35	409.58	129.94	79.64	68.77	22.86	332.11	26 out	29 out	22 out	30 out	25 out	31 out	**0.77**
mdd-p5	25	22 out	23 out	13 out	14 out	13 out	1 out	14 out	16 out	17 out	17 out	19 out	17 out	**73.06**
mdd-p7	9	95.46	68.41	24.73	20.38	23.28	6.82	21.98	44.68	50.12	55.29	39.54	39.58	**1.82**
mdd-p9	10	2.16	0.59	0.19	0.11	0.35	0.18	0.27	0.73	1.08	0.76	0.48	0.46	**0.06**
#Sol	114	57	91	101	100	101	113	74	53	53	38	42	33	**114**
Itime	114	3.96	2.69	1.72	1.64	1.62	1.79	2.21	2.61	2.89	3.03	3.30	3.43	2.09

Table 10.6: XCSP benchmarks.

the mdd-p5 and mdd-p7 series.



(a) Virtual best comparison **(b)** BCT-MS vs MDD-Min **(c)** The number of clauses

Figure 10.4: XCSP benchmarks.

Figure 10.4a shows that VB-Abscon and VB-BCT is the best method on around 90% and 10% instances, respectively. In addition, VB-BCT can be faster than VB-MDD on more than 80% instances. Figure 10.4b shows the differences between instances, PS is faster than Min on almost all instances in the bdd-15 and bdd-18 series but the opposite happens on the mdd-p5 and mdd-p7 instances, where Min is the best CNF encoding of MDD constraints for these instances. From Figure 10.4c, we can see that the number of clauses of the PS encoding can be 2-5 times less than that of Min on the bdd-15 and bdd-18 series.

We summarize experiments on all four benchmark series. While there is some initialization and encoding time for all methods, this is overall less significant than the solving time (there are many timeouts for some methods). The initialization time becomes significant when the encoding becomes large, e.g., in the NFA instances, the encoding cost becomes significant in the MDD CNF encodings with some being memory-out. Overall across all four problem

series, BCT CNF encodings generally outperform MDD CNF encodings. As with the MDD CNF encoding experiments in [AGMES16] where they found performance was mixed between CNF encodings and their propagator comparison, we also find that for some problems the BCT CNF encoding is the best while for other problems the BCT propagator in Abscon is the best. Still BCT CNF encoding is overall competitive or best for many instances and increases the flexibility and choices in solving of BCT (and NFA/MDD) constraints.

10.6 Conclusion

We investigate CNF encodings on BCT constraints which allow solving of BCT constraints with SAT solvers. At the same time, we show this can improve CNF encodings of MDD constraints. We tailor three well-known CNF encodings of binary constraints, i.e., the log encoding, direct encoding and support encoding, to encode BCT constraints. Then we propose two new CNF encodings, partial support encoding and minimal support encoding, which give smaller CNF encodings of BCT constraints. We study and compare the strength of unit propagation on these five CNF encodings of BCT constraints. Our experimental results study our CNF encodings of BCT constraints and also compare with seven existing CNF encodings of MDD constraints on a range of existing benchmarks. Experimental results show that the CNF encodings of BCT constraints can outperform those of MDD constraints. Our results show that solving of BCT constraints as well as NFA/MDD constraints is promising on SAT solvers.

Chapter 11

Conclusion

Typically non-binary constraints are handled using GAC propagators. However, another possibility is to transform the non-binary constraints with binary encodings into binary constraints, and then handling the binary constraints with AC propagators. It has been believed that the binary encoding approach while theoretically possible is not considered practically efficient. Thus, a "folklore" is that AC propagators through binary encodings cannot compete with GAC propagators for non-binary constraints. This is also why there has been a substantial body of work on many GAC propagators and their associated algorithms.

In this thesis, we show that binary encoding methods can work very well on various ad-hoc constraints, such as the table and Ordered Multi-valued Decision Diagram (MDD) constraints, which means the "folklore" is misleading. We propose new binary encodings to transform some ad-hoc constraints into binary constraints, and design specialized AC propagators to enforce AC on the binary encodings. In addition to AC propagators, we also tailor CNF encodings of binary constraints to transform binary encodings into CNF forms, making them suitable for SAT solvers. We remark that the binary encodings are only used to encode some ad-hoc constraints in a CSP model as binary constraints, while the other non-binary constraints in the CSP model are still handled with non-binary methods.

In Chapter 5, we show that specialized AC algorithms with existing binary encodings can be competitive with the state-of-the-art GAC algorithms of table constraints. We first give experimental results to explain why existing AC propagators on binary encodings are worse than table GAC propagators on the original non-binary constraints. We show that AC on HVE instances, generated by the well known binary encoding HVE, can be improved with a more efficient AC propagator, called HTAC, which exploits properties of the HVE encoding and a more flexible strategy of using search heuristics. The HTAC algorithm uses the special structure of the HVE encoding and various efficient techniques from state-of-the-art table GAC algorithms. Moreover, by using the HVE encoding with HTAC, we are able to avoid that the HVE encoding itself affects the search heuristic and consequently changes the search space compared with the original CSP. This effect can be quite significant, as we show in the experimental results. Our experimental results show that HTAC on the binary encoded instance is competitive with state-of-the-art table GAC

algorithms, in some cases, HTAC is faster. We believe the results reported in Chapter 5 are significant as it indicates that a long line of research algorithms with table GAC algorithms may not be needed or are less significant than the experimental results shown in those works [Ull07, CY10, Lec11, LLY12, MVHD12, XY13, JN13, PR14, GHLR14, WXYL16, DHL+16, VLS17, VLDS17, VLS18] suggest.[1]

In Chapter 6, we propose a new binary encoding called Bipartite Encoding (BE) to transform non-binary table constraints into binary constraints. The BE encoding partitions non-binary constraints into factor variables, and then representing the constraints as binary constraints between the factor variables. The binary constraints may share some factor variables, therefore, AC with the BE encodings can achieve high-order consistences on the original non-binary constraints. In addition, we also give an algorithm to generate compact bipartite encodings for non-binary table constraints. Furthermore, we present a specialized AC propagator to enforce AC on the binary constraints of the BE encoding exploiting their special structure. Our experiments on a large set of benchmarks using various search heuristics, such as the WDeg, Activity and Impact heuristics, show that the BE encoding with the specialized AC propagator can overall outperform both the state-of-the-art table GAC algorithms and binary encodings (the HVE encoding with HTAC). As far as we are aware, the most recent binary encoding of table constraints was proposed in 1999 (the double encoding [SW99b]). The HVE encoding proposed 30 years ago is still the state-of-the-art binary encoding used to transform table constraints into binary constraints. We propose a new encoding, i.e., the BE encoding, which we show to be state-of-the-art, thus, it is the first significant binary encoding after almost 20 years of absence.

In Chapter 7, we introduce an ad-hoc constraint representation called Binary Constraint Tree (BCT) which extends the representation introduced in [Dec87, Dec90a] by allowing hidden variables, where a BCT is a set of binary constraints with a tree structure. Furthermore, a binary encoding called direct tree binary encoding (DTBE) is proposed to transform non-binary decision diagram and automaton constraints into BCT constraints. Theoretically, the BCT constraints can be super-polynomially smaller than the MDD and NFA constraints. However, the BCT constraints generated from the DTBE encoding have a similar size to the original constraints. Therefore, we also give 4 reduction rules to simplify the BCT constraints by eliminating, merging and reconstructing hidden variables (introduced in Chapter 8). Additionally, we prove it is NP-hard to construct the minimal BCT constraints, and then giving a heuristic-based algorithm to simplify the DTBE encoding by use of the 4 reduction rules. Preliminary experimental results show that the simplified DTBE encoding can be much smaller than (up to 2166 times smaller than) the original DTBE encoding.

Then in Chapter 9, we present a specialized AC propagator to enforce AC on BCT

[1] See more details of table GAC propagators in Chapter 3.

constraints, which exploits the special tree structure of BCT constraints. Our experiments on a large set of benchmarks show that the AC propagator on BCT constraints can significantly outperform the state-of-the-art GAC propagators of the table and MDD constraints. To the best of our knowledge, the DTBE encoding proposed in the thesis is the first binary encoding used to transform decision diagram and automaton constraints into binary constraints. More importantly, the BCT constraint can be regarded as a new ad-hoc constraint which is more succinct than the MDD, regular and NFA constraints.

Finally, SAT solvers are an important and successful alternative for solving CSPs by encoding them into CNF forms, thus, we also try to make binary encodings suitable for SAT solvers. In Chapter 10, we investigate how to transform BCT constraints into a number of CNF forms. A BCT is a set of binary constraints, thus, the three well-known CNF encodings of binary constraints, i.e., the log encoding, direct encoding and support encoding, can also be directly used to encode BCT constraints. Furthermore, by exploiting the special tree structure of BCT constraints, we introduce two more compact specialized CNF encodings of BCT constraints. Additionally, we analyze and compare the strength of unit propagation on the five CNF encodings. Our experimental results show that the CNF encodings of BCT constraints can outperform existing MDD constraint CNF encodings on various benchmarks, confirming that binary encoding methods can also work well with the CNF encodings of binary constraints.

References

[AFNP14] Jérôme Amilhastre, Héléne Fargier, Alexandre Niveau, and Cédric Pralet. Compiling CSPs: A complexity map of (non-deterministic) multivalued decision diagrams. *International Journal on Artificial Intelligence Tools*, 23(04):1460015, 2014.

[AGMES16] Ignasi Abío, Graeme Gange, Valentin Mayer-Eichberger, and Peter J. Stuckey. On CNF encodings of decision diagrams. In *International Conference on Integration of AI and OR Techniques in Constraint Programming*, pages 1–17, 2016.

[AHM+08] Eric Allender, Lisa Hellerstein, Paul McCabe, Toniann Pitassi, and Michael Saks. Minimizing disjunctive normal form formulas and AC0 circuits given a truth table. *SIAM Journal on Computing*, 38(1):63–84, 2008.

[All83] James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[ALM20] Gilles Audemard, Christophe Lecoutre, and Mehdi Maamar. Segmented tables: An efficient modeling tool for constraint reasoning. In *European Conference on Artificial Intelligence*, pages 315–322, 2020.

[ANO+12] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for pseudo-boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012.

[AS14] Ignasi Abío and Peter J. Stuckey. Encoding linear constraints into SAT. In *International Conference on Principles and Practice of Constraint Programming*, pages 75–91, 2014.

[BB04] Nicolas Barnier and Pascal Brisset. Graph coloring for air traffic flow management. *Annals of operations research*, 130(1):163–178, 2004.

[BC93] Christian Bessière and Marie-Odile Cordier. Arc-consistency and arc-consistency again. In *AAAI National Conference on Artificial Intelligence*, 1993.

[BC94a] Nicolas Beldiceanu and Evelyne Contejean. Introducing global constraints in chip. *Mathematical and computer Modelling*, 20(12):97–123, 1994.

[BC94b] Christian Bessiere and Marie-Odile Cordier. Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1):179–190, 1994.

[BCP04] Nicolas Beldiceanu, Mats Carlsson, and Thierry Petit. Deriving filtering algorithms from constraint checkers. In *International Conference on Principles and Practice of Constraint Programming*, pages 107–122, 2004.

[BCR10] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog, 2010.

[BCSV17] Miquel Bofill, Jordi Coll, Josep Suy, and Mateu Villaret. Compact MDDs for pseudo-boolean constraints with at-most-one relations in resource-constrained scheduling problems. In *International Joint Conference on Artificial Intelligence*, pages 555–562, 2017.

[BCVBW02] Fahiem Bacchus, Xinguang Chen, Peter Van Beek, and Toby Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1-2):1–37, 2002.

[Ber95] Pierre Berlandier. Improving domain filtering using restricted path consistency. In *Conference on Artificial Intelligence for Applications*, pages 32–37, 1995.

[Bes99] Christian Bessiere. Non-binary constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 24–27, 1999.

[Bes06] Christian Bessiere. Constraint propagation. In *Foundations of Artificial Intelligence*, volume 2, pages 29–83. Elsevier, 2006.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[BFR95] Christian Bessiere, Eugene C. Freuder, and Jean-Charles Regin. Using inference to reduce arc consistency computation. In *International Joint Conference on Artificial Intelligence*, pages 592–598. Morgan Kaufmann Publishers Inc., 1995.

[BFR99] Christian Bessiere, Eugene C. Freuder, and Jean-Charles Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148, 1999.

[BHL04] Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. *International Workshop on Constraint Propagation and Implementation*, 4:29–43, 2004.

[BHLS04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *European Conference on Artificial Intelligence*, pages 146–150, 2004.

[BLAP16] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*, 2016.

[BLM15] Alessio Bonfietti, Michele Lombardi, and Michela Milano. Embedding decision trees and random forests in constraint programming. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 74–90, 2015.

[BMS12] Lucas Bordeaux and Joao Marques-Silva. Knowledge compilation with empowerment. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 612–624, 2012.

[BR97] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks: Preliminary results. In *International Joint Conference on Artificial Intelligence*, 1997.

[BR01] Christian Bessière and Jean-Charles Régin. Refining the basic constraint propagation algorithm. In *International Joint Conference on Artificial Intelligence*, volume 1, pages 309–315, 2001.

[BRYZ05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[BSW08] Christian Bessiere, Kostas Stergiou, and Toby Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172(6-7):800–822, 2008.

[BT93] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–69, 1993.

[BVB98] Fahiem Bacchus and Peter Van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 310–318, 1998.

[CJ98] Assef Chmeiss and Philippe Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(02):121–142, 1998.

[Coo89] Martin C. Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41(1):89–95, 1989.

[CWA+88] Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. *SICStus Prolog user's manual*. Swedish Institute of Computer Science Kista, 1988.

[CXY12] Kenil C. K. Cheng, Wei Xia, and Roland H. C. Yap. Space-time tradeoffs for the regular constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 223–237, 2012.

[CY06a] Kenil C. K. Cheng and Roland H. C. Yap. Applying ad-hoc global constraints with the case constraint to still-life. *Constraints*, 11(2-3):91–114, 2006.

[CY06b] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In *European Conference on Artificial Intelligence*, pages 78–82, 2006.

[CY06c] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad-hoc n-ary boolean constraints. In *European Conference on Artificial Intelligence*, pages 78–82, 2006.

[CY08] Kenil C. K. Cheng and Roland H. C. Yap. Maintaining generalized arc consistency on ad hoc r-ary constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 509–523, 2008.

[CY10] Kenil Cheng and Roland H. C. Yap. An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15(2):265–304, 2010.

[DB97] Romuald Debruyne and Christian Bessiere. From restricted path consistency to max-restricted path consistency. In *International Conference on Principles and Practice of Constraint Programming*, pages 312–326, 1997.

[DB01] Romuald Debruyne and Christian Bessiere. Domain filtering consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

[DD87] Avi Dechter and Rina Dechter. Removing redundancies in constraint networks. In *AAAI National Conference on Artificial intelligence*, pages 105–109, 1987.

[Dec87] Rina Dechter. Decomposing an n-ary relation into a tree of binary relations. In Moshe Y. Vardi, editor, *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 185–189, 1987.

[Dec90a] Rina Dechter. Decomposing a relation into a tree of binary relations. *Journal of Computer and System Sciences*, 41(1):2–24, 1990.

[Dec90b] Rina Dechter. On the expressiveness of networks with hidden variables. In *AAAI National Conference on Artificial intelligence*, pages 556–562, 1990.

[Dec92] Rina Dechter. Constraint networks. *Encyclopedia of Artificial Intelligence*, pages 276–285, 1992.

[Dec99] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999.

[Dec03] Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[DHL+16] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-table: Efficiently filtering table constraints with reversible sparse bit-sets. In *International Conference on Principles and Practice of Constraint Programming*, pages 207–223, 2016.

[DK01] Minh Binh Do and Subbarao Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

[DM02] Elizabeth D Dolan and Jorge J Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[Doy79] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, 1979.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[DP87] R Dechter and J Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[DP88] Rina Dechter and Judea Pearl. Tree-clustering schemes for constraint-processing. In *AAAI National Conference on Artificial Intelligence*, pages 150–154, 1988.

[DP89] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[dSMSSL13] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-sets for domain implementation. In *CP Workshop on Techniques for Implementing Constraint Programming Systems*, pages 1–10, 2013.

[DV94] Alvaro Del Val. Tractable databases: How to make propositional unit resolution complete through compilation. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 551–561, 1994.

[DVH10] Yves Deville and Pascal Van Hentenryck. Domain consistency with forbidden values. In *International Conference on Principles and Practice of Constraint Programming*, pages 191–205, 2010.

[FE96] Eugene C. Freuder and Charles D. Elfe. Neighborhood inverse consistency preprocessing. In *AAAI National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 202–208, 1996.

[FH20] Armin Biere Katalin Fazekas Mathias Fleury and Maximilian Heisinger. CaDiCaL, KISSAT, PARACOOBA, PLINGELING and TREENGELING entering the SAT competition 2020. *SAT COMPETITION*, 2020:50, 2020.

[FHJ+08] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[Fre78] Eugene C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[Fre82] Eugene C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[FS90] Mark S. Fox and Norman Sadeh. Why is scheduling difficult? A CSP perspective. In *European Conference on Artificial Intelligence*, pages 754–767, 1990.

[Gas77] John Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *International Joint Conference on Artificial Intelligence*, page 457, 1977.

[Gen02] Ian P. Gent. Arc consistency in sat. In *European Conference on Artificial Intelligence*, volume 2, pages 121–125, 2002.

[GGP05] Marc Gravel, Caroline Gagne, and Wilson L Price. Review and comparison of three methods for the solution of the car sequencing problem. *Journal of the Operational Research Society*, 56(11):1287–1295, 2005.

[GHLR14] Nebras Gharbi, Fred Hemery, Christophe Lecoutre, and Olivier Roussel. Sliced table constraints: Combining compression and tabular reduction. In *International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming*, pages 120–135, 2014.

[GJMN07] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI National Conference on Artificial Intelligence*, volume 7, pages 191–197, 2007.

[GLMS20] David Gerault, Pascal Lafourcade, Marine Minier, and Christine Solnon. Computing aes related-key differential characteristics with constraint programming. *Artificial intelligence*, 278:103183, 2020.

[GMS16] David Gerault, Marine Minier, and Christine Solnon. Constraint programming models for chosen key differential cryptanalysis. In *International Conference on Principles and Practice of Constraint Programming*, pages 584–601, 2016.

[GN04] Ian P. Gent and Peter Nightingale. A new encoding of alldifferent into SAT. In *International Workshop on Modelling and Reformulating Constraint Satisfaction*, pages 95–110, 2004.

[GNNS15] Nina Ghooshchi, Majid Namazi, MA Hakim Newton, and Abdul Sattar. Transition constraints for parallel planning. In *AAAI National Conference on Artificial Intelligence*, 2015.

[GSS11] Graeme Gange, Peter J. Stuckey, and Radoslaw Szymanek. MDD propagators with explanation. *Constraints*, 16(4):407, 2011.

[GSW00] Ian Gent, Kostas Stergiou, and Toby Walsh. Decomposable constraints. *Artificial Intelligence*, 123(1-2):133–156, 2000.

[HE80] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[HPRS06] Willem-Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. Revisiting the sequence constraint. In *International Conference on Principles and Practice of Constraint Programming*, pages 620–634, 2006.

[HS17] Emmanuel Hebrard and Mohamed Siala. Explanation-based weighted degree. In *International Conference on Integration of AI and OR Techniques in Constraint Programming*, pages 167–175, 2017.

[IM94] Kazuo Iwama and Shuichi Miyazaki. SAT-variable complexity of hard combinatorial problems. In *In Proceedings of the World Computer Congress of the IFIP*. Citeseer, 1994.

[JJNV89] Philippe Janssen, Philippe Jégou, Bernard Nouguier, and Marie-Catherine Vilarem. A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation. In *IEEE International Workshop on Tools for Artificial Intelligence*, pages 420–427. Citeseer, 1989.

[JN13] Christopher Jefferson and Peter Nightingale. Extending simple tabular reduction with short supports. In *International Joint Conferences on Artificial Intelligence*, pages 573–579, 2013.

[JRL08] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an open source java constraint programming library. In *CPAIOR Workshop on Open-Source Software for Integer and Contraint Programming*, pages 1–10, 2008.

[Kas90] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

[KHR+02] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *AAAI National Conference on Artificial Intelligence*, 2002.

[KS19] Petr Kučera and Petr Savický. Propagation complete encodings of smooth DNNF theories. *CoRR*, abs/1909.06673, 2019.

[KS20] Petr Kučera and Petr Savický. Bounds on the size of PC and URC formulas. *Journal of Artificial Intelligence Research*, 69:1395–1420, 2020.

[Kum92] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–32, 1992.

[KW07] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 379–393, 2007.

[Lag08] Mikael Zayenz Lagerkvist. *Techniques for efficient constraint propagation*. PhD thesis, KTH, 2008.

[LBH03] Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *International Conference on Principles and Practice of Constraint Programming*, pages 480–494, 2003.

[Lec08] Christophe Lecoutre. Optimization of simple tabular reduction for table constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 128–143, 2008.

[Lec11] Christophe Lecoutre. STR2: optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371, 2011.

[LH07] Christophe Lecoutre and Fred Hemery. A study of residual supports in arc consistency. In *International Joint Conference on Artificial Intelligence*, pages 125–130, 2007.

[LLGL13] Hongbo Li, Yanchun Liang, Jinsong Guo, and Zhanshan Li. Making simple tabular reductionworks on negative table constraints. In *AAAI National Conference on Artificial Intelligence*, 2013.

[LLY12] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A path-optimal GAC algorithm for table constraints. In *European Conference on Artificial Intelligence*, pages 510–515, 2012.

[LLY15a] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. Improving the lower bound of simple tabular reduction. *Constraints*, 20(1):100–108, 2015.

[LLY15b] Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. STR3: A path-optimal filtering algorithm for table constraints. *Artificial Intelligence*, 220:1–27, 2015.

[LPS13] Christophe Lecoutre, Anastasia Paparrizou, and Kostas Stergiou. Extending STR to a higher-order consistency. In *AAAI National Conference on Artificial Intelligence*, 2013.

[LR05] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *AAAI National Conference on Artificial intelligence*, pages 405–410, 2005.

[LS06] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 284–298, 2006.

[LSTV07] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Nogood recording from restarts. In *International Joint Conference on Artificial Intelligence*, pages 131–136, 2007.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[LV08] Christophe Lecoutre and Julien Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2:21–35, 2008.

[LXY14] Chavalit Likitvivatanavong, Wei Xia, and Roland H. C. Yap. Higher-order consistencies through GAC on factor variables. In *International Conference on Principles and Practice of Constraint Programming*, pages 497–513, 2014.

[LXY15] Chavalit Likitvivatanavong, Wei Xia, and Roland H. C. Yap. Decomposition of the factor encoding for CSPs. In *International Joint Conference on Artificial Intelligence*, pages 353–359, 2015.

[LZBF04] Chavalit Likitvivatanavong, Yuanlin Zhang, James Bowen, and Eugene C. Freuder. Arc consistency in mac: a new perspective. *Constraint Propagation and Implementation*, 4:93–107, 2004.

[Mac77a] Alan K. Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[Mac77b] Alan K. Mackworth. *On reading sketch maps.* Department of Computer Science, University of British Columbia, 1977.

[Mai83] David Maier. *The theory of relational databases*, volume 11. Computer science press Rockville, 1983.

[McG79] James J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19(3):229–250, 1979.

[MDL15] Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. The smart table constraint. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 271–287, 2015.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. *Artificial intelligence*, 28(2):225–233, 1986.

[MLB01] Sylvain Merchez, Christophe Lecoutre, and Frédéric Boussemart. AbsCon: A prototype to solve CSPs with abstraction. In *International Conference on Principles and Practice of Constraint Programming*, pages 730–744, 2001.

[MM88] Roger Mohr and Gérald Masini. Good old discrete relaxation. In *European Conference on Artificial Intelligence*, pages 651–656, 1988.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132, 1974.

[MS01] Nikos Mamoulis and Kostas Stergiou. Solving non-binary CSPs using the hidden variable encoding. In *International Conference on Principles and Practice of Constraint Programming*, pages 168–182, 2001.

[MVH12] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, pages 228–243, 2012.

[MVHD12] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. An optimal filtering algorithm for table constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 496–511, 2012.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *International Conference on Principles and Practice of Constraint Programming*, pages 529–543, 2007.

[Orl77] James Orlin. Contentment in graph theory: covering graphs with cliques. In *Indagationes Mathematicae*, volume 80, pages 406–424. Elsevier, 1977.

[Pes04] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *International Conference on Principles and Practice of Constraint Programming*, pages 482–495, 2004.

[PR14] Guillaume Perez and Jean-Charles Régin. Improving GAC-4 for table and MDD constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 606–621, 2014.

[PR15] Guillaume Perez and Jean-Charles Régin. Efficient operations on MDDs for building constraint programming models. In *International Joint Conference on Artificial Intelligence*, pages 374–380, 2015.

[Pre09] Steven Prestwich. CNF encodings. *Handbook of satisfiability*, 185:75–97, 2009.

[QW06] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In *International conference on principles and practice of constraint programming*, pages 751–755, 2006.

[Raz16] Igor Razgon. On the read-once property of branching programs and cnfs of bounded treewidth. *Algorithmica*, 75(2):277–294, 2016.

[Ref04] Philippe Refalo. Impact-based search strategies for constraint programming. In *International Conference on Principles and Practice of Constraint Programming*, pages 557–571, 2004.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI National Conference on Artificial Intelligence*, 1994.

[Rég96] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *AAAI National Conference on Artificial Intelligence*, pages 209–215, 1996.

[Rég11a] Jean-Charles Régin. Global constraints: A survey. In *Hybrid optimization*, pages 63–134. Springer, 2011.

[Rég11b] Jean-Charles Régin. Improving the expressiveness of table constraints. In *International Workshop on Constraint Modelling and Reformulation*, 2011.

[RP97] Jean-Charles Régin and Jean-François Puget. A filtering algorithm for global sequencing constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 32–46, 1997.

[RPD90] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In *European Conference on Artificial Intelligence*, volume 90, pages 550–556, 1990.

[SC06] Christian Schulte and Mats Carlsson. Finite domain constraint programming systems. *Handbook of Constraint Programming*, page 493, 2006.

[SC18] Anthony Schneider and Berthe Y. Choueiry. PW-CT: extending Compact-Table to enforce pairwise consistency on table constraints. In *International Conference on Principles and Practice of Constraint Programming*, 2018.

[Sel06] Meinolf Sellmann. The theory of grammar constraints. In *International Conference on Principles and Practice of Constraint Programming*, pages 530–544, 2006.

[SF94] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *International Workshop on Principles and Practice of Constraint Programming*, pages 10–20, 1994.

[SG97] Barbara M Smith and Stuart A Grant. Trying harder to fail first. In *European Conference on Artificial Intelligence*, pages 249–253, 1997.

[SHMB90] Arvind Srinivasan, Timothy Ham, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 92–95, 1990.

[SS77] Richard M Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[SS05] Nikolaos Samaras and Kostas Stergiou. Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *Journal of Artificial Intelligence Research*, 24:641–684, 2005.

[SSW00] Barbara Smith, Kostas Stergiou, and Toby Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *AAAI National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 182–187, 2000.

[STL13] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with gecode. *Web site: http://www. gecode. org*, 2013.

[SW99a] Kostas Stergiou and Toby Walsh. The difference all-difference makes. In *International Joint Conference on Artificial Intelligence*, volume 99, pages 414–419, 1999.

[SW99b] Kostas Stergiou and Toby Walsh. Encodings of non-binary constraint satisfaction problems. In *AAAI National Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 163–168, 1999.

[Tsa87] Edward PK Tsang. The consistent labeling problem in temporal reasoning. In *AAAI National Conference on Artificial Intelligence*, pages 251–255, 1987.

[Ull07] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Sciences*, 177:3639–3678, 2007.

[VFA99] Mathieu Veron, Hélène Fargier, and Michel Aldanondo. From CSP to configuration problems. In *AAAI Workshop on Configuration*, pages 18–19, 1999.

[VG08] Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008.

[VHDT92] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.

[VK86] Marc Vilain and Henry Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI National Conference on Artificial Intelligence*, volume 86, pages 377–382, 1986.

[VLDS17] Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extending compact-table to basic smart tables. In *International Conference on Principles and Practice of Constraint Programming*, pages 297–307, 2017.

[VLS17] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-table to negative and short tables. In *AAAI National Conference on Artificial intelligence*, pages 3951–3957, 2017.

[VLS18] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-MDD: Efficiently filtering (s)MDD constraints with reversible sparse bit-sets. In *International Joint Conference on Artificial Intelligence*, 2018.

[VLS19] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Extending compact-diagram to basic smart multi-valued variable diagrams. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 581–598, 2019.

[VP18] Julien Vion and Sylvain Piechowiak. From MDD to BDD and arc consistency. *Constraints*, 23(4):451–480, 2018.

[Wal93] Richard J Wallace. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In *International Joint Conference on Artificial Intelligence*, volume 93, pages 239–245, 1993.

[Wal99] Toby Walsh. Search in a small world. In *International Joint Conference on Artificial Intelligence*, pages 1172–1177, 1999.

[Wal00] Toby Walsh. SAT v CSP. In *CP 2008*, pages 441–456, 2000.

[WLPT19] Hugues Wattez, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Refining constraint weighting. In *International Conference on Tools with Artificial Intelligence*, pages 71–77, 2019.

[WXY17] Ruiwei Wang, Wei Xia, and Roland H. C. Yap. Correlation heuristics for constraint programming. In *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 1037–1041. IEEE, 2017.

[WXYL16] Ruiwei Wang, Wei Xia, Roland H. C. Yap, and Zhanshan Li. Optimizing simple tabular reduction with a bitwise representation. In *International Joint Conference on Artificial Intelligence*, pages 787–795, 2016.

[XY13] Wei Xia and Roland H. C. Yap. Optimizing STR algorithms with tuple compression. In *International Conference on Principles and Practice of Constraint Programming*, pages 724–732, 2013.

[ZY01] Yuanlin Zhang and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *International Joint Conference on Artificial Intelligence*, volume 1, pages 316–321, 2001.