NEW RELATIONAL CONSISTENCY ALGORITHMS AND A FLEXIBLE SOLVER ARCHITECTURE FOR INTEGRATING THEM DURING SEARCH

by

Anthony R. Schneider

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Doctor of Philosophy

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry

Lincoln, Nebraska

December, 2022

NEW RELATIONAL CONSISTENCY ALGORITHMS AND A FLEXIBLE SOLVER ARCHITECTURE FOR INTEGRATING THEM DURING SEARCH

Anthony R. Schneider, Ph.D.

University of Nebraska, 2022

Adviser: B.Y. Choueiry

Consistency algorithms, which perform inference, are at the heart of Constraint Programming. The strongest consistency level provided in most constraint solvers is Generalized Arc Consistency (GAC). In recent years, higher-level consistencies, especially relational consistencies, were shown to be critical for solving difficult Constraint Satisfaction Problems (CSPs). Implementing algorithms that enforce such consistencies in existing solvers cannot be done in a flexible and transparent manner and may require significantly modifying the constraint model of the CSP.

In this thesis, we address the practical mechanics for making higher-level consistencies a pragmatic choice for solving CSPs. To this end, we present three main contributions. First, we design and implement a new generation of constraint solvers with an architecture open to development and integration of new domain and relation-filtering consistency algorithms where one or more consistency algorithms can independently operate on specific, possibly overlapping subproblems, and where the implementation of the relational consistency algorithms does not require the modification of the constraint model of the CSP. Second, we propose new algorithms for two different kinds of relational consistencies: pairwise and m-wise consistency. Finally, we describe how to dynamically identify and exploit tractable substructures during search using one of the novel pairwise consistencies developed in this dissertation.

ACKNOWLEDGMENTS

I would like to extend my heartfelt thanks to my advisor, Professor Berthe Y. Choueiry, for her guidance, instruction, and patience throughout my time in graduate school. I also need to thank the other members of the Constraint Systems Laboratory who contributed to STAMPEDE and availed themselves to me for countless brainstorming sessions: Robert Woodward, Daniel Geschwender, Ian Howell, Denis Komissarov, and Nate Stender.

I would be remiss not to mention at least some of the friends I made during my tenure at the University of Nebraska-Lincoln, without whom I would likely not have seen this through to the end: Robert Woodward, Daniel Geschwender, Taylor Spangler, Jake Williams, Ellie Quint, Ian Howell, Mikaela Cashman, Natasha Pavlovikj, Nancy Pham, Dan Gutierrez, and so many others. Without their friendships, my journey through academia would have been considerably worsened. You all helped me grow and gave me the confidence to push on through all the years here.

And of course, my sincerest thanks to my amazingly loving family, and wonderful fiancée Lakyn for enduring me during challenging times and encouraging me through it all.

This research was partially supported by NSF Grant No. RI-111795 and NSF Grant No. RI-1619344. Experiments were conducted on the equipment of the Holland Computing Center at the University of Nebraska–Lincoln.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint Processing (CP) is a powerful framework for modeling and solving a wide range of practical problems including scheduling problems, resource allocation, and product configuration. The flexibility of the framework lends itself to models that are more easily produced and interpreted by humans, especially when compared to many of CP's closely related approaches such as SAT Solvers or Mathematical Programming

In general, CP optimization problems are $\mathcal{NP}$-Hard and their decision variants, called Constraint Satisfaction Problems (CSPs), are $\mathcal{NP}$-Complete. A CSP is defined by a set of decision variables with their respective set of values (i.e., domain) and a set of constraints that restrict the acceptable combinations of values per variable. A solution to a CSP is an assignment of values to all variables such that all constraints are satisfied. In many applications, we seek one solution to a CSP.

Mechanisms for solving a CSP typically intertwine two main mechanisms, namely, search and inference. The work presented in this dissertation uses constructive backtrack search, a sound and complete algorithm that exhaustively explores the search space to find a solution to a CSP. Inference is used to narrow the search space by pruning portions of the search space that cannot contain a satisfying solution. The focus of this dissertation are CSPs, the relational consistency algorithms used for inference, the exploitation of the changing topology of the graphical structure of a CSP instance during search, and the design and implementation of a flexible solver architecture to support the integration of diverse consistency algorithms.

1.1 Motivation & Claim

Backtrack search is currently the only sound and complete algorithm for solving CSPs. It is a deterministic and exhaustive process. However, its cost grows exponentially in the size of the problem. In order to limit the exponential growth of the cost of search, inference algorithms are used before and during search. They operate on subproblems to remove inconsistent values or combinations of values, which results in pruning the search space and removing subtrees that do not contain any solution. Inference procedures (also called *consistency propagators* and *consistency algorithms*) enforce a specific *consistency property* and are typically polynomial in time and space in the size of the problem. Their efficiency stems from the fact that they operate locally on subproblems of a restricted size and propagate their effects over the entire problem. The larger the subproblems they consider, the stronger the consistency they enforce, but also the larger the computational cost.

The strongest consistency property enforced by common (i.e., off-the-shelf) constraint solvers is Generalized Arc Consistency (GAC) [Mackworth, 1977; Waltz, 1975]. GAC focuses on a *single* constraint at a time, ensuring that the values in the domains of the variables to which the constraint applies do not violate the constraint. In contrast, the research community has investigated propagators that reason about a *combination* of constraints (e.g., algorithms for Path Consistency [Montanari, 1974] or more general relational consistencies [Dechter and van Beek, 1997; Karakashian *et al.*, 2010a]). They typically enforce stronger consistency properties than GAC and are thus called *high-level consistencies* (HLC). Existing constraint solvers cannot easily accommodate these more complex inference algorithms, because propagators in most existing solvers are typically tied to a single constraint. This situation is perhaps best exemplified by the following quote from the tutorial of the CHOCO constraint solver: [1]

> When one needs to declare its own constraint, actually, he needs to create a propagator. Indeed, in CHOCO, a constraint is a container which is composed of propagators.

We believe that this limitation of modern solvers has hindered the progress of research on relational consistency algorithms.

Although the cost of relational consistency algorithms may seem too high to be practical, during the past decade, a few researchers, including some in the Constraint Systems Laboratory, have designed HLC algorithms and established their usefulness in practical settings [Karakashian *et al.*, 2010a, 2012, 2013; Karakashian, 2013; Lecoutre *et al.*, 2013; Paparrizou and Stergiou, 2012, 2016; Samaras and Stergiou, 2005; Schneider and Choueiry, 2018; Schneider *et al.*, 2014; Stergiou, 2007; Woodward *et al.*, 2011a,b,c, 2012]. While relational consistency propagators can potentially prune a much larger amount of the search space than traditional propagators, the time required to enforce relational consistencies can often be prohibitive in practice. In fact, traditional propagators (e.g., GAC) frequently outperform existing relational consistencies when the relational consistencies are indiscriminately applied at every variable assignment during search. This situation prompts the following question:

How can relational consistency algorithms become a pragmatic choice for solving CSPs?

In this dissertation, we answer the above question as follows:

1. We propose and implement a new constraint-solver architecture that allows the simultaneous application of various domain and relation-filtering

[1]https://choco-tuto.readthedocs.io/en/latest/src/801.constraints.html

consistency algorithms on selectively chosen and potentially overlapping subproblems.

2. We design new relational consistency algorithms that not only improve upon existing ones, but are competitive with the fastest available traditional consistency algorithms.

3. We advantageously exploit the dynamically changing topology of the graphical representation of the CSP in order to reduce the search effort.

4. Finally, we empirically show that combining HLC algorithms and the exploitation of structural properties as enabled by our new solver is advantageous in practice.

As a result, this dissertation establishes the promise and practicality of higher-level consistencies in constraint solvers.

1.2 Approach

Our approach to addressing the challenge of furthering the promise of relational consistencies focuses on two complementary aspects of constraint processing: the architecture of solvers and the theoretical and empirical evaluation of relational and traditional consistency algorithms.

1.2.1 Rethinking Constraint Solvers

We believe that there are three aspects of the architecture of a constraint solver that are critical for making higher-level consistencies a competitive and pragmatic choice, namely,

1. The predictability of a propagator's behavior,
2. Fine-tuned control over queues and ordering, and
3. The ease of implementation and integration of consistency algorithms in the solver.

We will show that Stampede, the solver developed in this dissertation, achieves these goals by inverting the traditional relationship between constraints and their propagators adopted by most common constraint solvers. More specifically, rather than a constraint being associated with multiple propagators, Stampede allows propagators to directly operate over an arbitrary set of constraints. This architecture facilitates the development of relational consistency algorithms and the ability to flexibly and selectively apply different types of consistencies during search over specific subproblems. Further, allowing developers of propagators to fully control the order of queueing and handling of the ordering of operations within subproblems facilitates the empirical evaluation of consistency algorithms by limiting the variability encountered in existing solvers.

1.2.2 A New Generation of Relational Consistency Algorithms

We advance the current state of the art in relational consistencies through the creation of four new algorithms for enforcing the consistency properties (f)PWC and R($*$,m)C, namely, PerFB [Schneider *et al.*, 2014], AllSolFB, PW-AC2, and PW-CT [Schneider and Choueiry, 2018].

As a first step towards advancing relational consistencies to be ready for practical use, we propose PerFB, a consistency algorithm that enforces R($*$,m)C. PerFB is an improvement on a previous algorithm PerTuple [Karakashian *et al.*, 2010a;

Karakashian, 2013]. We identify and exploit a weakness in the original algorithm that caused redundant calls to PerTuple's backtrack search.

The algorithm PW-AC of Samaras and Stergiou [2005] enforces PWC on a CSP without modifying the constraint network and uses the piecewise-functional property of the constraints of the dual CSP to efficiently propagate tuple deletions by deleting all the tuples in an equivalence class instead of one at a time. However, it creates data structures for *each pair* of connected constraints in the dual CSP, causing a large memory overhead in many problems, which poorly scale in the presence of large relations and dense graphs. Our algorithm, PW-AC2, improves upon PW-AC by exploiting the minimality of the dual graph, and operating on the *distinct* subscopes incident to constraints, instead of pairs of constraints in the dual graph. As a result, we can significantly reduce the costs of enforcing fPWC both in terms of time and memory requirements. Finally, we push the state-of-the-art in algorithms for enforcing PWC with PW-CT, an algorithm that is competitive with the (as of writing) fastest algorithm for enforcing GAC.

1.2.3 Dangle Identification: Identifying Opportunities For Selectively Applying Relational Consistencies

We propose to develop an efficient technique to dynamically identify α-acyclic portions of the subproblem induced after the instantiation of variables during search when solving non-binary CSPs. To this end, we adapt the *Graham reduction operator*, in combination with counting the degree of vertices in the dual graph in order to reduce cost.

As these *dangling*, acyclic branches are identified, we propose to use a modified version of our state-of-the-art PW-CT algorithm to ensure that each dangle has at least one solution *in polynomial time* thanks to well-known properties of tree-structured CSPs. If a dangle is found to be unsatisfiable, search has to backtrack. If it is found to be satisfiable, we propose to remove the tractable subproblem, thus reducing the number of variables over which search has to iterate and the number of constraints for inference algorithms to consider. Search can then process the remaining subproblem.

1.3 Contributions

Below are the primary contributions of this dissertation, each of which functions as a step towards our goal of demonstrating how relational consistencies can become a pragmatic choice for application in CSPs:

1. *A new solver which priorities research and exploration of HLC* We design and build, Stampede, the first constraint solver that allows us to execute any consistency algorithm (of any level) on any subproblem of a CSP. Algorithms can naturally and transparently operate on variable/value and relation/tuple encodings of the primal, dual, or incidence graphs *without* requiring any alteration of the constraint model. This feature allows the creation of hybrid propagators that selectively enforce, during search, arbitrary consistencies on arbitrary groups of variables and constraints. The flexibility and modularity of the current version of our solver, Stampede, has already been validated in two contexts. The first context is the implementation of reactive strategies that trigger, during search, high-level consistencies as required by the problem instance difficulty [Woodward, 2018]. The second context is a portfolio approach for controlling the execution of minimality algorithms on subproblems during search [Geschwender, 2018].

2. *Pushing the state-of-the-art for Pairwise Consistency algorithms* We introduce two new algorithms for enforcing Pairwise Consistency (PWC) [Samaras and Stergiou, 2005]. These algorithms dramatically improve performance by exploiting the piecewise functionality of the equality constraints in the dual network. We show that the performance of our latest algorithm for PWC is comparable to that of the state-of-the-art algorithm for Generalized Arc Consistency (GAC), which is the most commonly used algorithm in research constraint solvers and enforces the strongest consistency in public-domain constraint solvers.

3. *Improved algorithms for enforcing m-wise consistency* We improve upon the existing state-of-the-art algorithms for enforcing m-wise consistency, PerTuple [Karakashian *et al.*, 2010b] and AllSol [Karakashian, 2013], by identifying and skipping redundant work when enforcing the consistency.

4. *Dynamically exploiting the changing structure of the search space* We design new mechanisms for dynamically identifying tractable subproblems during search based on the changing topology of the constraint network. Further, we advocate efficiently (i.e., in polynomial time) solving these substructures by partially enforcing PWC.

We also present in this thesis the following secondary contributions, which are not directly related to our goal but are nevertheless significant:

1. *Live data Visualizations in* Stampede We implement a mechanism for visualizing arbitrary algorithms, data structures, and search itself in Stampede while search is running. This mechanism has already been used in research material to identify new ways of intelligently applying varying levels of consistency during search [Woodward, 2018].

2. *Live debugging of Search and Consistency algorithms* Using the same tools that enabled live data visualization, we integrate a live debugger into STAMPEDE that allows setting break points in the code that can pause search, allowing a user to easily examine the full search state. This mechanism was integral in debugging and analyzing many of the algorithms presented in this dissertation.

1.4 Outline of Dissertation

The remainder of this dissertation is organized as follows:

- **Chapter 2** provides the background information necessary to contextualize the contributions of this thesis.
- **Chapter 3** details the architecture and aspects of STAMPEDE which make it novel and integral to our research into relational consistencies.
- **Chapter 4** provides two new algorithms for enforcing PWC. Results from this chapter appeared in [Schneider and Choueiry, 2018].
- **Chapter 5** provides two new algorithms for enforcing m-wise consistency. Results from this chapter appeared in [Schneider *et al.*, 2014].
- **Chapter 6** introduces a mechanism for dynamically identifying tractable subproblems of a CSP during search.
- **Chapter 7** Concludes this dissertation and highlights interesting areas for future work that arise from the research presented here.

Summary

This chapter provided an overview of our motivations, claims, and road-path to verifying those claims, as well as an outline of the remainder of this dissertation.

Chapter 2

Background

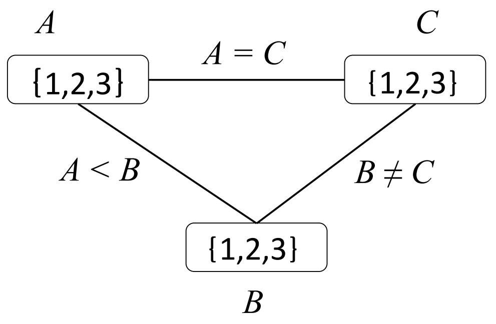
This chapter provides information necessary to contextualize the research presented in this dissertation. We then review the research related to consistency algorithms used in this dissertation and present an overview of the most popular and modern CSP solvers.

2.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined as $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, where

- $\mathcal{X}$ is a set of variables
- $\mathcal{D}$ the set of the variables' domain values, with $x_i \in \mathcal{X}$ having the domain $d_i \in \mathcal{D}$, and
- $\mathcal{C}$ is a set of constraints $c_i = \langle R_i, \mathit{scope}(c_i)\rangle$, where $\mathit{scope}(c_i) \subseteq \mathcal{X}$, and $R_i = \{(a, b, ..., c)\} \subseteq \prod_{v_i \in \mathit{scope}(c_i)} d_i$.

The *arity* of a constraint is the cardinality of its scope. Solving a CSP requires assigning to each variable a value from its domain such that all the constraints are satisfied. We frequently refer to the *neighbors* of a variable x_i as the set of variables that appear in the scope of any constraint c_j where $x_i \in \mathit{scope}(c_j)$. In this dissertation, we consider variables with finite domains and a specific type of constraint called a *table constraint*, where relations are given by *tuples*, which are the valid combinations of values for the variables in the constraint's scope. We use the relational projection operator π to restrict a tuple to a set of variables.



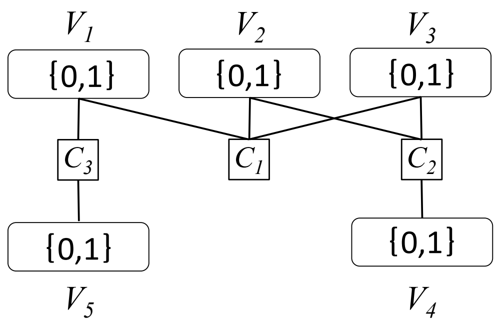
A	**B**
1	2
1	3
2	3

A	**C**
1	1
2	2
3	3

B	**C**
1	2
1	3
2	1
2	3
3	1
3	2

Figure 2.1: A CSP with intension constraints and its corresponding table constraints.

Figure 2.1 shows a small example CSP with three variables (A, B, and C), each represented by a node in a graph. Each variable has the domain $\{1, 2, 3\}$. The constraints are represented by edges between the nodes, which are labeled with a mathematical expression corresponding to the constraint between the two variables. Mathematically formulated constraints such as those appearing on the edges in the graph of Figure 2.1 are called *intension* constraints. The intension and table constraints shown in the figure are equivalent.



C_1

V_1	V_2	V_3
0	0	1
0	1	0
1	0	0
0	1	1
1	0	1
1	1	0
1	1	1

C_2

V_2	V_3	V_4
0	0	1
0	1	0
1	0	0
0	1	1
1	0	1
1	1	0
1	1	1

C_3

V_1	V_5
0	0
0	1
1	1

Figure 2.2: SAT clauses represented as a CSP.

As a further example, consider the Boolean expression in Conjunctive Normal Form: $(V_1 \vee V_2 \vee V_3) \wedge (V_2 \vee V_3 \vee V_4) \wedge (\neg V_1 \vee V_5)$. An equivalent CSP is shown in

Figure 2.2. Each term in the SAT is represented as a variable in its equivalent CSP, and each clause is a constraint. The resulting CSP is *non-binary* because the arity of at least one of the constraints is greater than two.

2.1.1 Solving CSPs

To date, backtrack search remains the only sound and complete algorithm for finding a solution to a CSP. Below, we review three important components of a search procedure used for solving CSPs: variable ordering heuristics, inference procedures for pruning the search space, and the search procedure itself.

Variable Ordering Heuristic: Currently, the most effective variable ordering heuristic is the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ heuristic, which chooses the variable with the minimal value obtained from dividing the cardinality of its remaining domain with the sum of the weighted degrees of its incident edges [Boussemart *et al.*, 2004]. The weight of a constraint is incremented when the constraint causes a domain wipeout of a variable, causing the search to backtrack. This heuristic adheres to the principle of instantiating the most constrained variable first. This principle allows us to reduce the branching factor of the search tree as well as to increase the pruning effectiveness of the filtering mechanisms resulting from inference. Older heuristics such as $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ are sometimes still used in research due to their relative stability with respect to the search space explored by different algorithms [Woodward *et al.*, 2011b, 2012]. The $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ heuristic operates exactly as $\frac{|\texttt{dom}|}{\texttt{wdeg}}$, but assigns the weight of each constraint to one.

Inference: Inference procedures enforce a given consistency property and (in general) operate by removing, from the domains of the uninstantiated variables in the problem, values that are guaranteed to be inconsistent with the instantiated variables. For our purposes, we adopt a *Real-Full Lookahead* (RFL) strategy [Nadel, 1989], which maintains a given consistency property over the subproblem induced by the uninstantiated variables given the set of current instantiations (i.e., given a partial solution).

Backtrack Search: The search procedure proceeds in a systematic and exhaustive manner generating consistent partial solutions. It assigns a value to the variable selected by the ordering heuristic and enforces some form of lookahead on the remaining subproblem given a new variable assignment. If the lookahead procedure empties the domain of a 'future' variable, we say the domain of the future variable is *annihilated*, and search proceeds by assigning another value to the variable. If all values for the current variable yield a dead-end, then backtracking occurs, which undoes the assignment of the previous variable, and the process is repeated until all variables are instantiated or no solution is found. In this dissertation, we are concerned with finding the first solution to the CSP.

Tree-Structured CSPs and Dangles: Tree structured CSPs can be solved backtrack-free (and thus, in polynomial time) by making the problem directional arc-consistent from the root to the leaves of the tree [Freuder, 1982b]. Performing a second pass of directional arc-consistent from the leaves to the root ensures the graph is minimal [Dechter, 2003a]. The first pass from root to leaves is sufficient for determining the existence of a solution. Notably, this mechanism applies to any *subproblem* of the CSP that is a tree. We say that such tree-structured subproblems form *dangles* attached to the constraint graph of the CSP. If we can identify and isolate them, we can determine their satisfiability *independently* of the rest of the CSP. One of the contributions of this dissertation is to dynamically (i.e., during search) identify such tractable substructures and exploit them to reduce the search effort.

R_1

A	B	C	E
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	1	0
0	1	1	1
1	1	1	1

R_2

A	B	D
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

R_3

A	B	C	G
0	0	1	0
0	0	0	1
0	1	1	0

R_4

A	C	F
0	0	1
1	0	1
1	1	0

Figure 2.3: The relations of a small CSP with satisfying tuples for a solution highlighted.

Solution Example: A valid solution to a CSP is one such that all variables are assigned, and none of the assignments causes a constraint to be violated. Consider a CSP with variables $\mathcal{X} = A, B, C, D, E, F, G$, each of which has a *boolean* domain (that is, $\forall x_i \in \mathcal{X}, d_i = 0, 1$), and constraints $\mathcal{C} = c_1, c_2, c_3, c_4$ where $\mathit{scope}(c_1) = A, B, C, E$, $\mathit{scope}(c_2) = A, B, D$, $\mathit{scope}(c_3) = A, B, C, G$, and $\mathit{scope}(c_4) = A, C, F$. The relations for each of these constraints is given in Figure 2.3. One valid solution to this example (there are many) is $\langle A, 0\rangle$, $\langle B, 0\rangle$, $\langle C, 0\rangle$, $\langle D, 0\rangle$, $\langle E, 1\rangle$, $\langle F, 1\rangle$, $\langle G, 1\rangle$. The highlighted tuples in Figure 2.3 are the tuples that satisfy these variable assignments.

2.1.2 Graphical Representations

There are several graphical representations for a CSP. In the *hypergraph*, the vertices represent the variables of the CSP and the hyper edges represent the scopes of the constraints. The two most common visualizations of a hypergraph are presented in Figure 2.4.

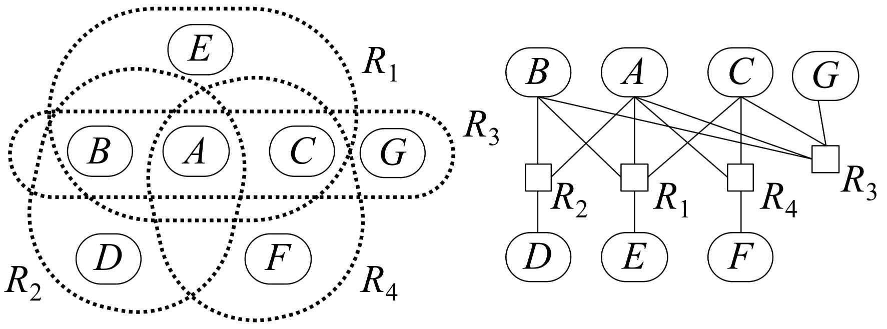


Figure 2.4: Two representations of a hypergraph

In the *primal graph*, the variables are vertices. Two vertices in the primal graph are connected by an edge when at least one constraint is shared amongst those two variables (Figure 2.5).

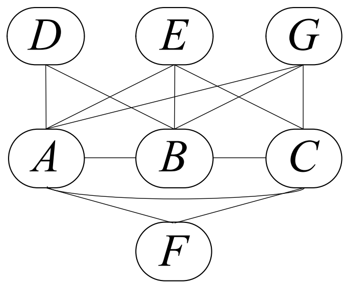
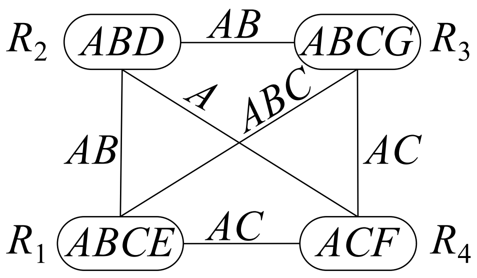


Figure 2.5: The primal graph of the hypergraph in Figure 2.4



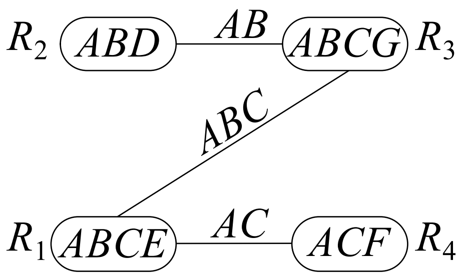


Figure 2.6: The dual graph representation of the graph in Figure 2.4

Figure 2.7: A minimal dual graph of the graph in Figure 2.4

In the *dual graph* (a.k.a. *join graph* [Dechter, 2003b] and complete intersection graph [Maier, 1983]), the vertices represent the constraints of the CSP and an edge exists between any two vertices whose scopes overlap (Figure 2.6). We call this non-empty intersection a *subscope*, which we denote, for two CSP constraints c_i and c_j, $subscope(c_i, c_j) = scope(c_i) \cap scope(c_j)$. This edge represents an equality constraint between c_i, c_j and indicates that the variables in $subscope(c_i, c_j)$ must take the same values in c_i and c_j (a.k.a. the *connectedness* property).

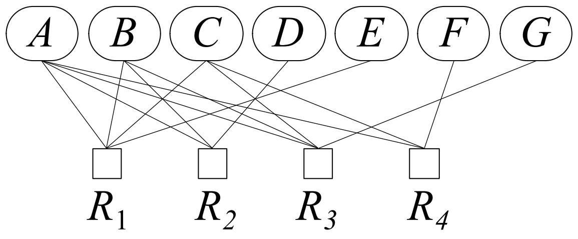


Figure 2.8: The incidence graph of the hypergraph in Figure 2.4

The *incidence graph* of a CSP is a bipartite graph, with the variables in the CSP forming the first set of vertices and the constraints in the CSP forming the second. An edge is present between a vertex representing the variables and one representing the constraints *iff* the variable is in the scope of the constraint. An example of an incidence graph is shown in Figure 2.8.

2.1.3 Properties of the Dual Graph

In this dissertation, we exploit two properties of the constraints of the dual graph of a CSP, namely, acyclicity and piecewise functionality.

Minimal Dual Graph: Janssen *et al.* [1989] and Dechter [2003b] observed that an edge between two vertices in the dual graph is *redundant* if there exists an alternative path between the two vertices such that the shared variables appear in every vertex in the path. Redundant edges can be removed without affecting the set of solutions (Fig. 2.7). Consequently, if any minimal dual graph of a CSP is a tree (a.k.a. *join* tree), then all its minimal dual graphs are necessarily trees. Janssen *et al.* [1989] introduced an efficient algorithm for computing the *minimal dual graph* by removing redundant edges. Many minimal graphs may exist, but all are guaranteed to have the same number of edges.

Acyclicity: Acyclicity of both the dual and hypergraph is an important component of this dissertation. There are several types of acyclicity defined for a given graphical model. [1] The strongest type of acyclicity, and the one we use in this dissertation, is α-acyclicity. A hypergraph is α-acyclic *iff* it has a join tree [Beeri *et al.*, 1983]. The properties of being α-acyclic, having a join tree, and being solvable with pairwise consistency are all equivalent properties of a query schema [Beeri *et al.*, 1983]. The Graham reduction is a graph reduction operator used to determine if a query schema (or, in our case, a hypergraph) is α-acyclic [Maier, 1983].

Piecewise Functionality: Another important property of the dual graph arises from the nature of the constraints in the dual encoding of a CSP. In Figure 2.9, the subscopes AB and AC label two edges and each of the subscopes A and ABC label one edge. These constraints are equality constraints and dictate that the variables of the subscope must have the same values in the corresponding vertices.

A binary constraint is said to be piecewise functional if the domains of the variables in its scope can be partitioned such that a set from one variable is supported by at most one set in the other and vice versa. Equality constraints are, therefore, *piecewise functional* [Hentenryck *et al.*, 1992; Samaras and Stergiou, 2005]. This property is critical in developing the new relational consistency algorithms presented in this dissertation.

[1] A quick review: Berge acyclicity [Beeri *et al.*, 1983] $\rightarrow$ γ-acyclicity [Duris, 2012] $\rightarrow$ β-acyclicity [Duris, 2012] $\rightarrow$ α–acyclicity [Beeri *et al.*, 1983] $\equiv$ join tree [Beeri *et al.*, 1983].

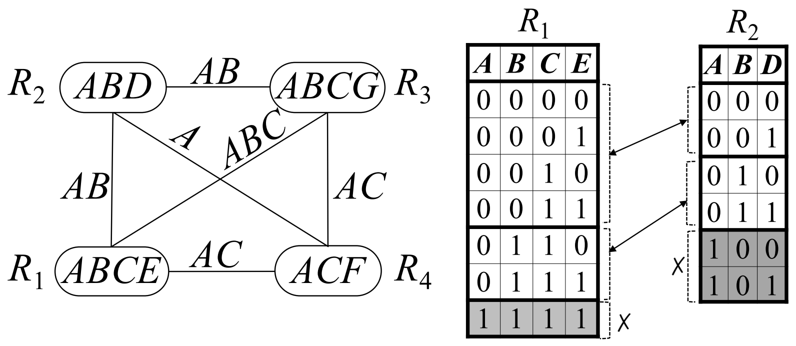


Figure 2.9: The equality constraint is piecewise functional

2.1.4 Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of its constraint network. We call the nodes in this tree its *clusters*. A given cluster *cl* is comprised of a set of variables $\chi(cl) \in \mathcal{X}$, and a set of constraints $\psi(cl) \in \mathcal{C}$. All valid tree decompositions must satisfy the following two conditions:

1. Each constraint appears in at least one cluster, and at least one of these clusters must also contain all the variables in the constraint's scope

2. For all variables $x_i \in \mathcal{X}$, the clusters where x_i appear induce a connected subtree of the decomposition

Many methods of constructing a tree decomposition from a CSP exist. STAMPEDE's current implementation, and the one used for tree decompositions in this thesis, is based on a technique presented in Dechter and Pearl [1989], and operates as follows:

1. We *triangulate*[2] the primal graph using the MINFILL heuristic [Dechter, 2003c; Kjærulff, 1990].

[2]A graph is triangulated *iff* every cycle of size four or more has a *chord*, which is an edge between any two non-adjacent vertices.

2. We compute the maximal cliques in the resultant triangulated primal using the MaxCliques algorithm [Golumbic, 1980].

3. Each of the identified cliques becomes a tree cluster cl in the final tree decomposition, where the variables in the clique compose the variables $\chi(cl)$.

4. A constraint c_i is placed in a cluster *iff* $scope(c_i) \subseteq \chi(cl)$. This may cause some clusters to have no constraints, which is acceptable as long as the two conditions listed above are satisfied.

5. The clusters are connected using the JoinTree algorithm [Dechter, 2003d], choosing a root for the tree such that the tree depth is minimized.

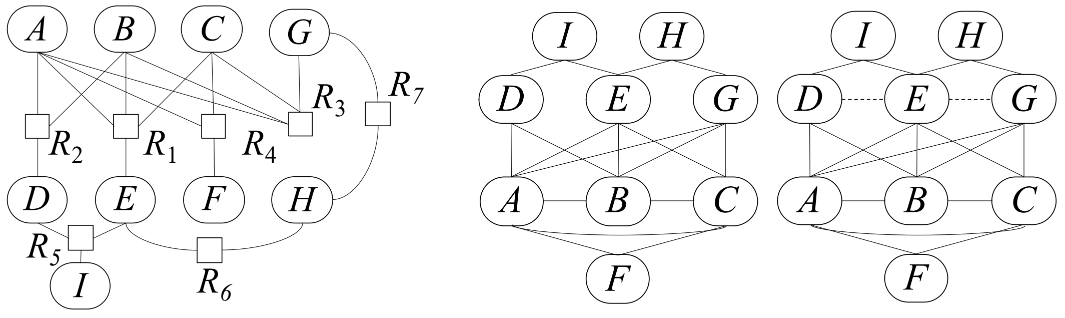


Figure 2.10: A hypergraph, its primal graph, and the triangulated primal graph

Figure 2.10 shows a hypergraph of a CSP[3], its corresponding primal graph, and the triangulated primal graph (with dashed lines representing the edges added by the triangulation). The maximal cliques for this CSP are shown in Figure 2.11, where each of the colored regions map to a maximal clique. Finally, the tree decomposition generated from the JoinTree algorithm is shown in Figure 2.12.

While not a central focus of this dissertation, tree decompositions have been used extensively to improve relational consistencies [Geschwender *et al.*, 2016; Geschwen-

[3]A slightly modified hypergraph from those presented earlier to make the resulting tree decomposition slightly more interesting.

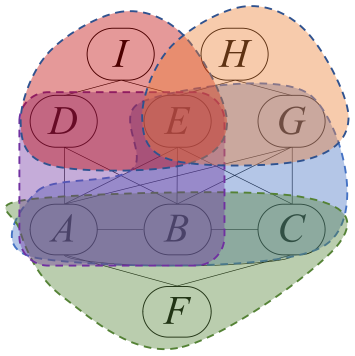


Figure 2.11: The max cliques for the problem in Figure 2.10

$$\begin{tikzcd}
 & \{A,B,C,E,G\}, \{R_1,R_3\} \arrow[dl] \arrow[d] \arrow[dr] & \\
\{E,G,H\}, \{R_5,R_6\} & \{A,C,F\}, \{R_4\} & \{A,C,D,E\}, \{R_2\} \arrow[d] \\
 & & \{D,E,I\}, \{R_7\}
\end{tikzcd}$$

Figure 2.12: The tree decomposition for the graphs in Figure 2.10

der, 2018; Karakashian *et al.*, 2011, 2012, 2013; Karakashian, 2013; Woodward, 2018]. Most relevant to this work is their use in conjunction with the propagation algorithms PerTuple and AllSol to focus the enforcement of m-wise consistency over every combination of m relations in each cluster, rather than every combination of m relations in the entire problem, which can drastically reduce the memory and time requirements of the algorithm.

2.2 Consistency Properties and Algorithms

Consistency properties can be partitioned into global and local properties. While global consistency properties are likely intractable, algorithms for enforcing local consistency properties are typically polynomial time because they operate on subproblems of fixed sizes. For a given consistency property, there can be any number of algorithms (i.e., propagators) for enforcing it on a CSP. Additionally, consistency properties can also be partitioned into those that focus on the domains of the variables and those that focus on the relations of the constraints. This dissertation deals primarily with three local consistency properties, namely, *Generalized Arc Consistency* (GAC), *Pairwise Consistency* (PWC), and m-wise consistency (R($*$,m)C). Below, we review their definitions.

2.2.1 Global Consistency Properties

Minimality and decomposability [Montanari, 1974] are both highly desirable *global consistency properties*, which is a consistency property defined over an entire CSP. Minimality ensures that every value in the domain of the variables participate in at least one solution. Decomposability means that any partial solution of size k can be extended to a complete solution backtrack-free. Unfortunately, enforcing minimality is $\mathcal{NP}$-Complete [Gottlob, 2011], and generally speaking global consistency properties are computationally hard [Bessiere, 2006]. For this reason, the remainder of this dissertation focuses on specific examples of *local consistency properties* and the algorithms that enforce them.

2.2.2 Generalized Arc Consistency

Arc Consistency is a simple variable-based consistency property that operates on binary CSPs and, when enforced, guarantees every value in every variable has at least one support in each of its neighbors. Generalized Arc Consistency (GAC) is a version of Arc Consistency that can be applied to non-binary CSPs. Both Arc Consistency and GAC are extremely popular consistency properties due to their low cost to enforce during search.

Definition 1. Generalized Arc Consistency (GAC) [Mackworth, 1977; Waltz, 1975]: *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is GAC* iff*, for every constraint $c_i \in \mathcal{C}$, and $\forall x_j \in scope(c_i)$, every value $v \in D(x_j)$ is consistent with c_i (i.e., appears in some support of c_i).*

There are many general algorithms for enforcing GAC on arbitrary constraints. Algorithms that strictly enforce GAC only remove unsupported values from the domains of relations, but an extension to GAC algorithms (referred to as *Tabular Reduction*) also filter tuples from the relations of constraints which can no longer be used in a solution. The current state of the art algorithm for GAC (and the one used in this dissertation as the baseline for empirical evaluations) is a tabular reduction algorithm called CompactTable(CT) [Demeulenaere *et al.*, 2016]. This algorithm makes heavy use of a data structure called the *reversible sparse bitset*, which is similar to the *sparse set structure* [Briggs and Torczon, 1993; le Clément *et al.*, 2013], but encodes each element of the set as a single bit. Sparse sets are optimized for membership checks as well as for insertions and removal of values. Making a sparse set reversible allows the restoration of elements in the set in $O(1)$ time and space, enabling extremely fast backtracking operations.

2.2.3 Pairwise Consistency

Pairwise consistency is a relation-based consistency that effectively enforces arc consistency on the dual encoding of a CSP.

Definition 2. Pairwise Consistency (PWC) [Gyssens, 1986]: *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is PWC* iff*, for every tuple t_i in every constraint c_i there is a tuple*

t_j in every constraint c_j such that $\pi_{subscope(c_i,c_j)}(t_i) = \pi_{subscope(c_i,c_j)}(t_j)$, t_j is called a PW-support of t_i in c_j. A CSP that is both PWC and GAC is said to be full PWC (fPWC) [Debruyne and Bessière, 2001].

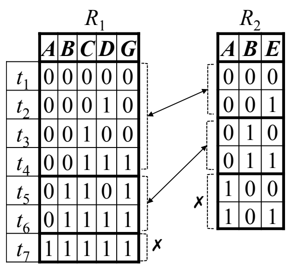
As mentioned in Section 2.1.2, the piecewise functionality property divides relations into equivalences classes of tuples (see Figure 2.13) and forms the basis of the PW-AC algorithm [Samaras and Stergiou, 2005]. PW-AC monitors the number of living tuples in each group of tuples formed by the equivalence classes induced by the piecewise functionality property. When one of the groups' count is reduced to 0, all other groups in the equivalence class are immediately removed from the problem. However, the memory required to store the equivalences classes induced by the piecewise functionality property is prohibitive in practice.

2.2.4 m-wise Relational Consistency

Pairwise consistency operates on every pair of relations in a CSP, which prompts the question, *could we enforce a relational consistency over an arbitrary number of relations*? Gyssens [1986] proposed a property for relational databases called m-wise consistency. This property ensures that every combination of m relations was minimal (that is, each tuple in a relation could be extended to every combination of $m-1$ relations). R($*$,m)C [Karakashian *et al.*, 2010a, 2013] is equivalent to the m-wise consistency property defined in Databases.

Definition 3. R($*$,m)C [Karakashian *et al.*, 2010a]: *A constraint network $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is R($*$,m)C* iff *every tuple in the relation of each constraint $c_i \in \mathcal{C}$ can be extended to the variables in $\bigcup_{c_j \in \mathcal{C}} scope(c_j) \setminus scope(c_i)$ in an assignment that satisfies all the constraints in $\mathcal{C}$ simultaneously. A network is R($*$,m)C* iff *every set of m constraints, $m \geq 2$, is R($*$,m)C.*

The parameterized algorithm PerTuple enforces R($*$,m)C. It ensures that each tuple in a relation appears in a solution of the dual CSP induced by the m relations by conducting a backtrack search on the tuples of the $m-1$ relations (see Figure 2.14). A closely related consistency is wR($*$,m)C, which guarantees R($*$,m)C for every set of m connected constraints found in the *minimal* dual graph.



R_1	A	B	C	D	G
t_1	0	0	0	0	0
t_2	0	0	0	1	0
t_3	0	0	1	0	0
t_4	0	0	1	1	1
t_5	0	1	1	0	1
t_6	0	1	1	1	1
t_7	1	1	1	1	1

A	B	E
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

Figure 2.13: Piecewise functional constraint.

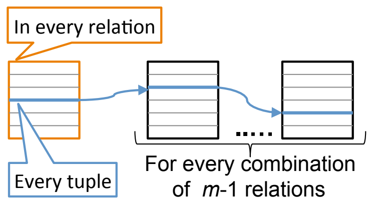


Figure 2.14: Illustrating R($*$,m)C.

After running PerTuple, the removal of tuples from a relation R_i are reflected in the domains of variables in $scope(R_i)$ by projecting the altered constraints onto the domains of the variables in the constraints' scopes. The PerTuple algorithm uses a data structure called an *index tree* that groups the equivalent tuples in a constraint relative to another constraint, implicitly exploiting the piecewise functionality of the two constraints. This data structure allows PerTuple's backtrack search to efficiently identify tuples that can be part of a valid solution to the combination of constraints.

2.3 Related Work

In this section, we delve into work related to our goal of demonstrating the efficacy and aiding in the development of relational consistencies.

2.3.1 Constraint Solvers

At a high level, constraint solvers fall into two categories: off-the-shelf solvers primarily used for modeling and industrial applications, and research solvers used for developing new consistency properties and algorithms. Many of these solvers straddle that line to some extent, but lean more towards one side or the other. There are three publicly available solvers that are commonly used for building commercial applications, namely, CHOCO, GECODE, and Google's OR-TOOLS.

CHOCO: CHOCO is a Java-based solver whose primary focus is to model and solve decision and optimization problems for commercial applications. It is maintained and used by Cosling to support their commercial activities. Of the three solvers discussed in this section, it is the most understandable and straightforward in its design and implementation, and the developers have provided ample documentation. While the current version of CHOCO is not intentionally built to support further research and algorithm development, the clarity of the code seems well suited for extensions. As stated above, CHOCO attaches propagators directly to constraints. This design decision raises no issues for commonly used consistency algorithms. However, the code may require significant alterations to lend itself to the development of relational consistency algorithms because the current solver's architecture does not support them.

GECODE: GECODE's architecture is a complex one, but, thankfully, GECODE comes with extensive documentation. It relies heavily on the visitor and signal design patterns, resulting in somewhat unpredictable behavior from a non-expert user's perspective. A scheduler class controls the execution order of propagators, which may not be transparent to the user. Like in CHOCO, propagators are associated with a single constraint. Thus, allowing a propagator to operate on multiple constraints would likely require altering the model of the CSP. If implementing relational consistencies is indeed possible, it is certainly not straightforward or trivial.

OR-Tools: OR-Tools is a library that combines, into one solver, graph algorithms, linear and mixed integer programming, constraint programming, and routing and scheduling algorithms. The constraint programming component of the solver focuses on optimization, not on satisfaction. Additionally, because of the breadth of its scope, the documentation of OR-Tools is fragmented, making it difficult to find a cohesive overview of the constraint solver. Recently, they have replaced the original constraint solver with a new CP-SAT solver, which reportedly boasts some performance improvements but, unfortunately,

> "The only (big) downside is that you cannot write new constraints with this solver."[4]

This limitation is indeed a significant obstacle to the development of new constraints and propagators and largely precludes the use of this solver for research.

We believe that all three solvers discussed above focus on modeling and solving practical applications rather than providing an open platform to support new research and investigations. However, given its design and extensive documentation, Choco may be, in our opinion, more accessible to a researcher than Gecode and OR-Tools. Furthermore, because of the automated behavior of their propagation schedulers, it is difficult to predict the behavior of these solvers without gaining an intimate knowledge of their control mechanisms. (This comment applies to all three solvers but more so to Gecode or OR-Tools than to Choco.) Such a design is clearly beneficial for the engineering of industrial solutions. However, it is to the detriment of the researchers

[4]Message from main developer of OR-Tools, Laurent Perron, on the Google groups forums (accessed 2021/05/14).

who need to reduce the number of independent variables when empirically comparing algorithms. While both CHOCO and OR-TOOLS explicitly state that they do not support relational consistency algorithms, nothing in GECODE's documentation lends credence to the contrary (at least without significant modification of either its code or the CSP's model).

One example of a publicly available solver that is oriented more towards the research community is AbsCon [Merchezn *et al.*, 2001]. Prior approaches such as Mairy *et al.* [2014] have tried to integrate HLC into AbsCon, but required adding new constraints in the problem, altering the model. Vion *et al.* [2011] proposed a more general method of integrating HLC into theoretically *any* event based solver, but again, this formulation required adding a global constraint that replaces the constraints the HLC would operate on, as well as introducing several other abstractions. Further, it does not alleviate our concerns with execution order. Without altering the propagation queue scheme or implementing some form of constraint prioritization, the solver may choose to uselessly enforce a weak consistency after a strong consistency. This approach provides a means of enforcing HLC in event-based solvers, but, in our view, significantly increases the complexity of reasoning about the model and operation of the solver, in addition to requiring somewhat significant alterations to the constraint network.

2.3.2 Consistency Algorithms

Traditional GAC propagators have recently been extended to filter not only the domains of variables but also the tuples of table constraints. The first and perhaps most well-known GAC propagators are the Simple Tabular Reduction (STR) family of algorithms [Lecoutre *et al.*, 2012; Lecoutre, 2011; Ullmann, 2007]. These algorithms have shown a significant performance improvement over other GAC algorithms at the times of their publications.

The current state of the art in GAC propagators is the CompactTable algorithm, which, in a manner similar to the STR algorithms, filters relations. The CompactTable algorithm operates on a bitset representation of a table constraint, which allows it to remove multiple tuples at once rather than a single tuple as in STR. This feature is advantageous both when checking whether or not a variable-value pair has a support in a constraint and when removing invalid tuples, as each operation can be performed in 64 tuple-chunks in a single operation on most modern CPU architectures. The algorithm demonstrates a tremendous performance improvement over STR algorithms and is, to the best of our knowledge, the best general-use GAC propagator that currently exists [Demeulenaere *et al.*, 2016].

Algorithms for enforcing stronger consistencies than GAC while filtering only the domains of the variables include algorithms for enforcing the following consistency properties: NIC [Freuder and Elfe, 1996], maxRPC [Debruyne and Bessière, 1997a], maxRPWC [Bessière *et al.*, 2008], SAC [Debruyne and Bessière, 1997b], NSAC [Wallace, 2015], and POAC [Bennaceur and Affane, 2001] and its adaptive version aPOAC [Balafrej *et al.*, 2014]. While all the above approaches demonstrate performance improvement over their contemporaneous GAC algorithms on specific benchmarks, none of them can beat GAC on all instances.

The very first relational-consistency algorithms (i.e., algorithms that operate on a combination of constraints) are perhaps those for enforcing *Path Consistency* (PC) on a binary CSP [Montanari, 1974]. More generally, the consistency of combinations of a given size for non-binary constraints was articulated as m-wise consistency by Gyssens [1986]. More recently, Samaras and Stergiou [2005] introduced the algorithm PW-AC, which enforces pairwise consistency (PWC or m-wise consistency where $m = 2$), by effectively ensuring arc-consistency on the dual representation of the CSP.[5] The PW-AC algorithm was a good step towards making relational consistencies competitive with traditional propagators on certain benchmarks, but the data structures were costly in terms of memory. Lecoutre *et al.* [2013] developed eSTR, which enforces PWC and is based on the STR algorithms. They also identified a special condition where GAC is sufficient for enforcing PWC, though, the memory required for its implementation remains prohibitive in many cases. Paparrizou and Stergiou [2016] generalized the approach used by eSTR into a suite of algorithms (the HOSTR algorithms) capable of enforcing a number of relational consistency properties that range in strength from stronger than GAC but weaker than fPWC to enforcing fPWC. They found that a weakened version of eSTR which does not requeue on tuple removals was superior in performance to enforcing fPWC with eSTR.

Enforcing what corresponds to m-wise consistency with m=2 and m=3 was shown to be beneficial in the context of the Minesweeper computer-game [Bayer *et al.*, 2006]. Karakashian *et al.* [2010a] and Karakashian [2013] proposed two general, parameterized algorithms for enforcing m-wise consistency (which they called R($*$,m)C) named PerTuple and AllSol. They also identified the importance of using a *minimal dual graph*, which was discussed in detail in Section 2.1.2. The performance of their algorithms is hindered by the large number of m-sized combinations in problems and its impact on the computational cost, although this cost can be compensated by the significant reduction of the search space and the search effort that R($*$,m)C may provide.

[5]The idea of enforcing PWC by enforcing AC on the dual CSP was first proposed by Janssen *et al.* [1989].

2.3.3 Structural Tractability

Freuder [1982a] showed that tree-structured CSPs are tractable and can be solved in a backtrack-free manner after enforcing arc consistency. He also linked the level of consistency that is needed to guarantee tractability to the *width* of the constraint graph of the CSP [Freuder, 1985]. This relationship is of significant theoretical importance, but its practical applicability is limited because enforcing consistency in this manner may require adding constraints to the constraint graph, which alters its width, thus, requiring a higher consistency level. This line of research was extended by Dechter and Pearl [1988] and Dechter and Pearl [1989] yielding algorithms that guarantee single-parameter tractability in terms of the *induced width* of the constraint networks. Beyond CSPs, Dechter [2003b] argues that these techniques are intimately related to elimination algorithms, dynamic-programming algorithms, join trees in databases, and tree decompositions in graph theory.

Exploiting tree decompositions for solving satisfiability, optimization, or model counting problems in CP or more general graphical models has received increased attention in the last two decades [Dechter *et al.*, 2001; Dechter, 1996, 1997; Jégou and Terrioux, 2003a; Jégou *et al.*, 2005; Kask *et al.*, 2005]. One CP technique that is particularly relevant to our work is the *cycle-cutset* method [Dechter and Pearl, 1987]. This technique identifies a cycle cutset of the constraint network of a binary CSP, which is a set of variables whose removal changes the remaining network into a tree. The algorithm iterates over finding a solution to the variables in the cutset using backtrack search, then extending the solution found to the remaining tree structure after enforcing directional arc-consistency. The cycle-cutset method reduces the exponential cost of solving the CSP to size of the cutset.[6]

[6]Finding the smallest cutset of a graph is an $\mathcal{NP}$-Hard problem.

Derenievicz and Silva [2018] focuses on a specific kind of acyclic structure in Numerical Constrained Global Optimization Problems that they dub “Epiphytic Trees”. According to their work, these trees are more common than Berge acyclic graphs, and can be solved backtrack free through a combination of GAC and relational arc consistency. However, relational arc consistency may require adding constraints to the model, and the strength of their approach seems limited to a very specific class of optimization problem. Further, they only check for their acyclic structures prior to search, and do not dynamically identify the appearance of these acyclic structures as variables are assigned.

To the best of our knowledge, there has been little work on dynamically identifying tractable subproblems during search. Examples include dangling trees in the dual graphs within PerTuple and AllSol [Karakashian *et al.*, 2010a; Karakashian, 2013], which have been significantly improved and carefully evaluated by Geschwender [2018]. This approach is limited to search on the dual graph.

Acyclic schemas in databases are highly desirable because they are known to be tractable. The Graham operator was proposed in relational databases to efficiently recognize acyclic queries [Maier, 1983]. We make extensive use of this operator to identify tractable subproblems *during* search.

Summary

In this chapter we provided a formal definition of a CSP, an overview of various techniques used for solving a CSP, and an example of such a solution . We then gave a brief review of various consistency properties and how they relate to one another, including the definitions of global, local, and relational consistencies. We ended with an overview of works related to the research presented in this dissertation with a focus on consistency algorithms and constraint solvers.

Chapter 3

Stampede: A CSP Solver Designed for Research, Extensibility, and Composability

Relational consistencies have not traditionally been at the forefront of research within the CP community; traditional variable-based and other lower order consistency properties have long dominated the discourse and research, largely due to their low performance overheads and ease of implementation. This has unfortunately but predictably meant that software created for both research and industrial use has not been conducive to the development of novel relational consistency properties.

This chapter describes Stampede, a CSP solver developed from the ground-up to facilitate the creation and research of relational consistency properties. We will motivate the creation of this solver, explain its central design philosophies, and describe (at a high level) the architecture that enables both traditional and relational consistency properties to be developed and tested with as little friction as possible.

3.1 Motivating the Creation of Stampede

There is a large upfront cost of developing a wholly new software stack. New software requires substantial planning, designing, and testing, on top of the obvious time investment of programming and debugging. It is difficult to justify this immense investment of time, especially when existing software seemingly accomplishes most of the goals that the new stack aspires to achieve. The process of creating Stampede was no exception: Not only were there existing CP solvers available that claimed to be versatile, powerful, and fast, but our own lab had a homegrown solver (SCSP) that was capable of developing new algorithms for relational consistencies (in fact, R($*$,m)C and RNIC were both initially developed using SCSP). Here we address why we chose to construct STAMPEDE rather than either modifying an existing solver to incorporate relational consistency algorithms or continuing to use and expand on SCSP.

3.1.1 The Shortcomings of Existing Alternatives

As mentioned in Section 2.3.1, adapting one of the existing solvers to incorporate relational consistencies such as GECODE or CHOCO was not just difficult, but likely impossible to do without altering the CSP model. While this may have changed in the intervening years since STAMPEDE was conceived, it would appear that there's still a tightly coupled relationship between a specific constraint and its propagator in both solvers. Both CHOCO's and GECODE's highest level of consistency enforced on table constraints is GAC, using the CT algorithm [Prud'homme, 2022b] [Schulte *et al.*, 2022].

Even were there an avenue to easily incorporate relational propagators into these solvers, there would still be a need to contend with existing code, developers, review processes, and licenses, which would further impede relational consistency research. Indeed, this could carry the advantage of having a built in user base which might encourage additional and further growth of relational consistencies, but that would be far from a guarantee. Developing in-house meant we could develop and iterate much faster by creating software tailor made to our skills and needs. Control in general was a secondary (but important) motivator; we could choose the language, tools, and ensure integration with our existing parsing and experimental tooling infrastructure.

Further, these solvers (and the supporting documentation) tend to focus more on modeling, search, and defining new, very specific constraints. There is less of an emphasis on creating or comparing new propagators for existing constraints and inference algorithms. Overall, it is our view that the most prominent constraint solvers are not designed with supporting research as a primary goal, least of all research on producing propagators for combinations of constraints.

3.1.2 SCSP: A Precursor to STAMPEDE

Prior to initiating the development of STAMPEDE, UNL's Constraint Systems Laboratory conducted research using another internally built constraint solver called SCSP. The development of SCSP began as a single student's effort, and quickly evolved into a full fledged solver to support the lab's research. However, this quick evolution resulted in a fragmented code base without a strong, cohesive design backing it, as elements were added in an ad-hoc manner to support new and changing research directions within the lab; the lab's investigations into relational consistencies were just beginning. Consequently, there was little-to-no documentation of code or design as elements were frequently removed, altered, or replaced to meet the lab's changing needs.

Further, SCSP was written using C99. The lack of modern programming features (e.g., objects, generics, lambdas) in combination with its rapid development led to a heavy reliance on global variables and global state. By the end of its use, a singleton global variable used ubiquitously throughout SCSP had grown to contain over one hundred individual members, many of which were pointers to other global structures. Managing state in SCSP had become an onerous task, and a non-trivial amount of development and research was spent debugging segmentation faults and initialization issues stemming from these global structures. The absence of a standard library in C99 also necessitated the development (and frequent debugging) of lab-grown data structures for everything ranging from linked lists to multiple graph models.

Configuration of SCSP was handled through command line arguments, using the `GetOpt` library. Predictably, as the number of algorithms and research directions grew, the command line arguments used to configure SCSP grew as well. Adding new options to the command line, either to support new algorithms or variations in existing ones, was an onerous process that typically required finding the correct parent switch, parsing it, ensuring it did not conflict with other switches, and passing it through several (sometimes a dozen or more) functions for use in the target algorithm. It was, by the end of SCSP's life-cycle, not uncommon to see command line runs such as `./scsp -f extra/xml/zebra-supports.xml -k -F -v-10 -Tmfrstonenosp -s2 -xpb -qmr -b2x13`, and even this is not a particularly egregious example.[1]

The point of relaying this here is not to tear down the efforts that went into building SCSP, but to clearly motivate the design decisions that informed STAMPEDE, as many hard lessons were learned that enabled us to design and build a solver that could streamline the development and testing of a wide range of research.

3.1.3 Design Philosophies and Goals of STAMPEDE

There is nothing particularly noteworthy about STAMPEDE with respect to the theoretical or research oriented field of software engineering; the aspect of STAMPEDE that makes it unique (and what this chapter will focus on) is how STAMPEDE treats the core entities of a CSP (e.g., constraints, relations, variables, and propagators) relative to its other contemporary constraint solvers.

[1]This specific configuration resulted in SCSP running an algorithm called PERTUPLE on a particular cluster in a tree decomposition (the 13th cluster, in this case).

Stampede is, in many ways, a response to the lessons learned from the design and implementation of SCSP and the shortcomings of other solvers with respect to their support for relational consistencies. Stampede was designed with a focus on the following objectives:

1. *Extensibility and Ease-of-use*: Stampede should be, above all else, easy to modify with as low of a barrier to entry for testing new ideas and algorithms as is feasible.

2. *Modularity and Composability*: Combining multiple approaches at once can produce surprising results. Propagators, ordering heuristics, and search type should be able to operate alongside each other without requiring "glue" for every new combination of algorithms. Propagators should be drop-in replacements for one another. And all without altering the initial model of the CSP.

3. *Research Oriented*: Making A/B comparisons between propagators should be as fair as possible. Propagators should share as much code as possible without sacrificing performance of the propagator itself. This ensures that differences seen between two algorithms can be attributed to the algorithms themselves.

Notably, Stampede was never intended to compete with other solvers. The performance of Stampede *relative to other solvers* was always an explicit non-goal; while Stampede needs to be fast enough to complete experiments, it was far more important to ensure that comparisons between two different propagators or algorithms were fair, all other things being equal. This required some sacrifice to overall performance, but care was paid to ensure that performance did not suffer so severely that we were unable to solve CSPs in a practical time frame. In short, Stampede tries to prioritize readability and usability over raw speed.

The remainder of this chapter will provide a high-level overview of how Stampede accomplishes these goals and expedites research and development of novel relational consistencies.

3.2 Extensibility: Lowering the Burden of Entry for Novel Ideas and Algorithms

As mentioned in Section 3.1.2, scsp inspired several quality-of-life improvements that became integral to the design of Stampede. In this section, we will describe how Stampede improved on the ease of adding of new algorithms through the use of a custom command-line argument parsing based on the "Templatized C++ Command Line Parser" (tclap) library [Aarno, 2022], then provide an overview of some of the core classes and relevant implementation details used to construct them, followed by describing the ways Stampede makes it straightforward to create a new propagator, and finally how propagator implementations can modify the state of a CSP without needing insight into the rest of the runtime environment.

3.2.1 Core Classes

Stampede provides a set of core classes used to represent a CSP. These classes are the building blocks for every search engine and constraint propagator. Here we'll provide a brief overview of some of these classes, focusing in particular on the elements that engender extensibility by providing a foundation on which new propagators can build upon.

As mentioned in Chapter 2, all CSPs are comprised of sets of variables, domains, and constraints. Unsurprisingly, these all have corresponding representations in Stampede. Each instantiation of one of these core classes has at least one unique

ID associated with them. Figure 3.1 shows a partial class hierarchy for one such class, the so-called `Assignable` class.

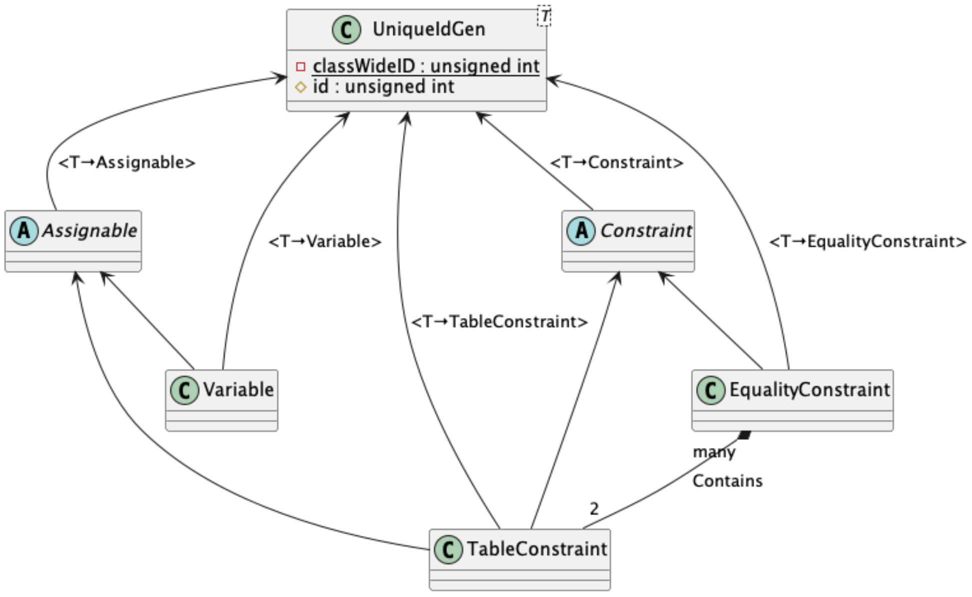


Figure 3.1: Relationship between subset of core classes

An `Assignable` represents any element which can be instantiated with a value. In traditional solvers, this would be likely be limited to variables and their domains, but Stampede implements several propagators which require temporary assignments of a *tuple* to a *table constraint*. An added benefit is that it closely reflects the nature of the dual encoding of a CSP.

Underlying the domains of the assignable objects in Stampede is the `RSparseIntrusiveDomain` class. This class is a generic container that combines two concepts: the *reversible sparse set* [Demeulenaere *et al.*, 2016] and an *intrusive list* [Boost, 2022]. A sparse set [Briggs and Torczon, 1993] is a data structure that acts as a set, but provides $O(1)$ complexity for insertion, removal, and lookup operations. A reversible sparse set is effectively the same structure, but adds a "reversible" primitive (essentially just a vector of values) that allows chunks of removed values to be restored in

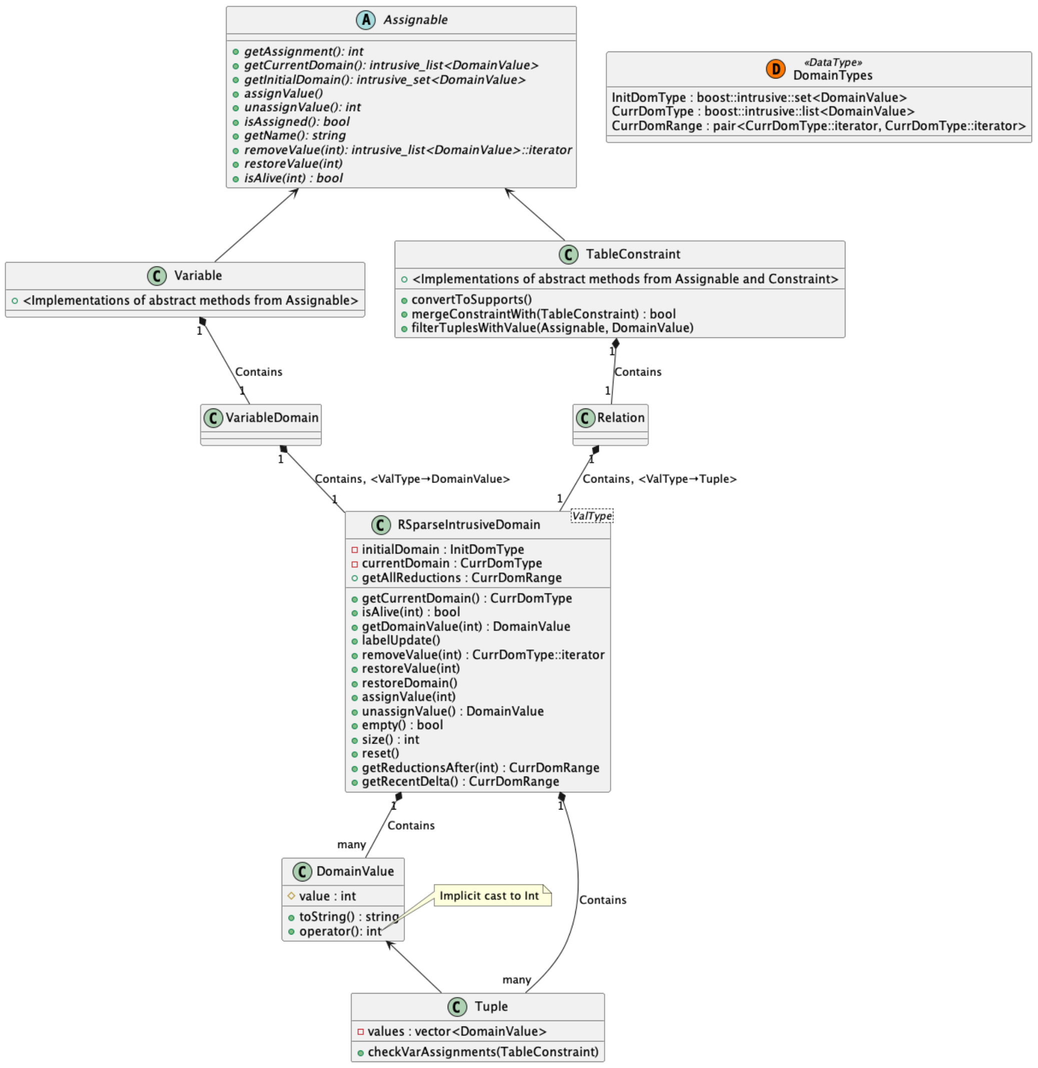

Assignable
getAssignment(): int
getCurrentDomain(): intrusive_list<DomainValue>
getInitialDomain(): intrusive_set<DomainValue>
assignValue()
unassignValue(): int
isAssigned(): bool
getName(): string
removeValue(int): intrusive_list<DomainValue>::iterator
restoreValue(int)
isAlive(int) : bool
«DataType»
DomainTypes
InitDomType : boost::intrusive::set<DomainValue>
CurrDomType : boost::intrusive::list<DomainValue>
CurrDomRange : pair<CurrDomType::iterator, CurrDomType::iterator>
Variable
<Implementations of abstract methods from Assignable>
TableConstraint
<Implementations of abstract methods from Assignable and Constraint>
convertToSupports()
mergeConstraintWith(TableConstraint) : bool
filterTuplesWithValue(Assignable, DomainValue)
1
Contains
1
1
Contains
1
VariableDomain
Relation
1
Contains, <ValType→DomainValue>
1
1
Contains, <ValType→Tuple>
1
RSparseIntrusiveDomain
ValType
initialDomain : InitDomType
currentDomain : CurrDomType
getAllReductions : CurrDomRange
getCurrentDomain() : CurrDomType
isAlive(int) : bool
getDomainValue(int) : DomainValue
labelUpdate()
removeValue(int) : CurrDomType::iterator
restoreValue(int)
restoreDomain()
assignValue(int)
unassignValue() : DomainValue
empty() : bool
size() : int
reset()
getReductionsAfter(int) : CurrDomRange
getRecentDelta() : CurrDomRange
1
Contains
many
1
Contains
many
DomainValue
value : int
toString() : string
operator(): int
Implicit cast to Int
Tuple
values : vector<DomainValue>
checkVarAssignments(TableConstraint)

Figure 3.2: Class diagram of STAMPEDE's assignable elements with a subset of class member variables and functions

constant time. The details of these data structures and their implementations have been covered extensively in prior literature, so are omitted from this dissertation.

An intrusive list is a cache-friendly implementation of a linked list where the underlying data can be stored in a compacted vector or array; pointers to the next element in the list are inserted into the objects themselves via a mix-in. This means all pointers to objects in the list remain valid for their lifetime (since the underlying objects are never moved), and slices of the list can be removed and re-inserted in $O(1)$ time as the only changes are pointer assignments. The combination of the intrusive list and reversible sparse set lets users of the domains iterate over either the living or dead values with value semantics (i.e., no pointers or `std::reference_wrappers`) while providing excellent performance when undoing changes during search or providing the changes made to the problem state to propagators and other objects.

Figure 3.2 shows the relationships between the `Variable`, `TableConstraint`, `VariableDomain`, and `Relation` classes. The `VariableDomain` and `Relation` classes are somewhat vestigial. With the introduction of `RSparseIntrusiveDomain`, both classes effectively just pass function calls and requests through to the underlying `RSparseIntrusiveDomain` implementation. In fact, a more efficient implementation of this structure would have a one-to-many relationship between domains and variables (or relations and table constraints) to save on space where possible, with an `RSparseIntrusiveDomain` representing the current domain.

Stampede also provides a small set of classes used to represent the various graphical representations of a CSP. Figure 3.3 shows an overview of these classes, but omits the classes representing the nodes and edges for space. The relationships of nodes and edges mirror the structure of the graphs, where dual edges and primal edges are instantiated with specific constraint types (equality and universal constraints, respectively).

The graphs are modeled using a pattern known colloquially as the *Curiously Recurring Template Pattern (CRTP)* [Coplien, 1995]. This allows a common base class, in this case `GenericGraph`, to implement a set of behaviors common to several child classes while allowing the child classes to specialize specific portions of the behavior implemented in the base class and add entirely new methods without the use of virtual functions. In the particular case of the `GenericGraph` class, we can offload

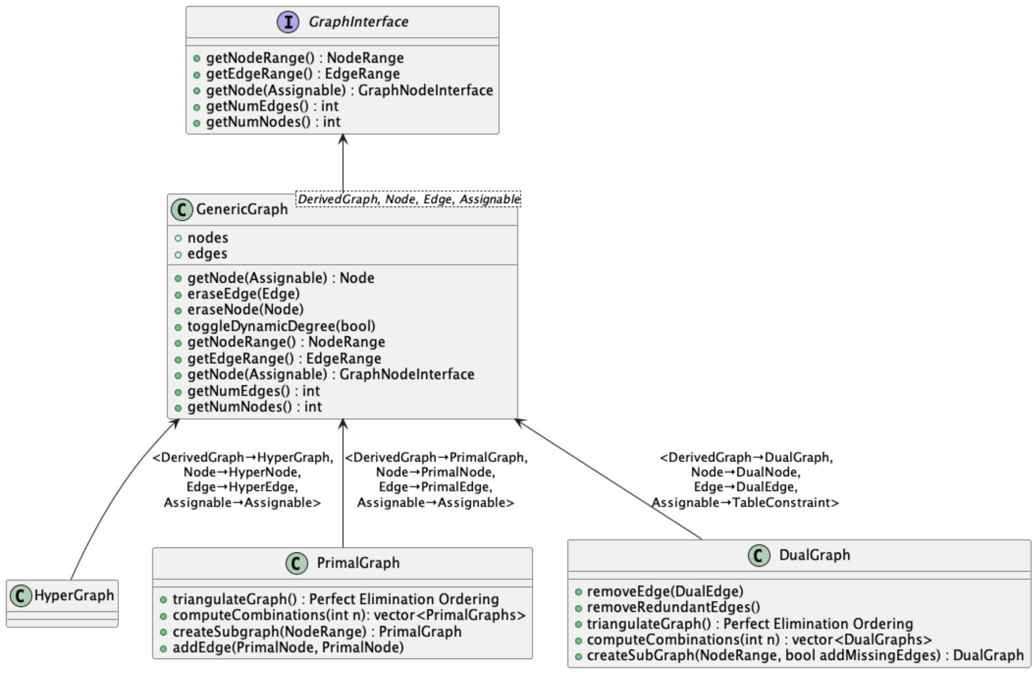

GraphInterface
getNodeRange() : NodeRange
getEdgeRange() : EdgeRange
getNode(Assignable) : GraphNodeInterface
getNumEdges() : int
getNumNodes() : int
GenericGraph
DerivedGraph, Node, Edge, Assignable
nodes
edges
getNode(Assignable) : Node
eraseEdge(Edge)
eraseNode(Node)
toggleDynamicDegree(bool)
getNodeRange() : NodeRange
getEdgeRange() : EdgeRange
getNode(Assignable) : GraphNodeInterface
getNumEdges() : int
getNumNodes() : int
<DerivedGraph→HyperGraph, Node→HyperNode, Edge→HyperEdge, Assignable→Assignable>
<DerivedGraph→PrimalGraph, Node→PrimalNode, Edge→PrimalEdge, Assignable→Assignable>
<DerivedGraph→DualGraph, Node→DualNode, Edge→DualEdge, Assignable→TableConstraint>
HyperGraph
PrimalGraph
triangulateGraph() : Perfect Elimination Ordering
computeCombinations(int n): vector<PrimalGraphs>
createSubgraph(NodeRange) : PrimalGraph
addEdge(PrimalNode, PrimalNode)
DualGraph
removeEdge(DualEdge)
removeRedundantEdges()
triangulateGraph() : Perfect Elimination Ordering
computeCombinations(int n) : vector<DualGraphs>
createSubGraph(NodeRange, bool addMissingEdges) : DualGraph

Figure 3.3: Class diagram of STAMPEDE's graph structures.

common behavior like adding or removing nodes and general graph construction so that the various child classes need not duplicate these implementation details.

The graph interface is seldom used in STAMPEDE. Most uses of any of the graph classes happen on a concrete instantiation of the class, because most algorithms know at compile time which representation they require (e.g., a specific propagator tends to require operating on a specific kind of graph, such as the dual graph). Generally speaking, the graph structures in STAMPEDE are lightweight, and allow developers to more easily reason across a multitude of CSP representations.

3.2.2 Run-time Configurable Algorithms: TCLAP and the CLI-Factory

Perhaps the aspect of STAMPEDE which had the largest impact on its ability to be easily modified and extended was its command-line parsing capabilities. One of the largest hurdles to deploying new algorithms (or modifications to existing algorithms) in SCSP was the large amount of boilerplate code and plumbing required to get started. This was remedied in STAMPEDE through the combination of two fairly simple methods: a generic factory class [Gamma *et al.*, 1994] called (`CLIFactory`), and a somewhat minor modification to an existing command-line parsing library called TCLAP [Aarno, 2022].

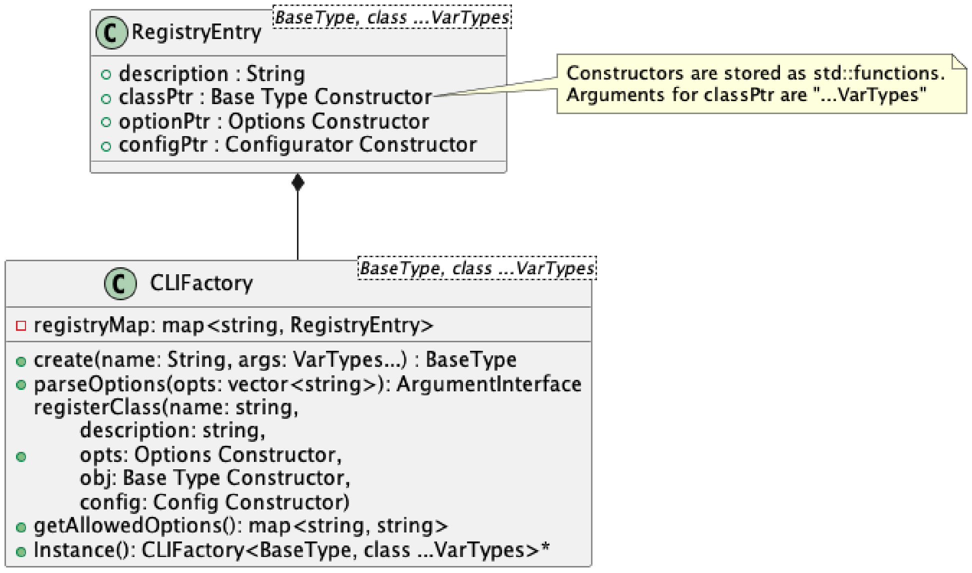

RegistryEntry
BaseType, class ...VarTypes
description : String
classPtr : Base Type Constructor
optionPtr : Options Constructor
configPtr : Configurator Constructor
Constructors are stored as std::functions. Arguments for classPtr are "...VarTypes"
CLIFactory
BaseType, class ...VarTypes
registryMap: map<string, RegistryEntry>
create(name: String, args: VarTypes...) : BaseType
parseOptions(opts: vector<string>): ArgumentInterface
registerClass(name: string, description: string, opts: Options Constructor, obj: Base Type Constructor, config: Config Constructor)
getAllowedOptions(): map<string, string>
Instance(): CLIFactory<BaseType, class ...VarTypes>*

Figure 3.4: A simplified UML diagram of STAMPEDE's CLI Factory

Figure 3.4 shows a simplified rendering of the `CLIFactory` class' API. The factory is a generic class, templated on the `BaseType` (e.g., a propagator or search engine), and a variable number of types that change depending on what `BaseType` the factory is being used to create. When registering a new class with the factory, a small number of arguments are required:

1. **Name:** The name of the class, which will be used to select the class for use on the command line.

2. **Description:** A short description of the class for use in help messages.

3. **Opts**: A function pointer or lambda to the constructor of the options used for the class.

4. **Obj:** A function pointer or lambda to the constructor of the class itself.

5. **Config:** A function pointer or lambda to the `Configurator` for the class.[2]

Once a class is registered with the factory, it can be created through the `create` method, which requires the name of the class to construct (supplied in the registration), and arguments corresponding to the same set of types used to define the factory.

This registration mechanism has a minor downside: the constructors for a given `BaseType` (e.g., all constructors for propagators) must have the same function definitions so that the signature of their lambda functions will be identical. This could potentially be worked around through type erasure or similar mechanisms, but was not necessary in the context of STAMPEDE: nearly all propagators, for example, rely on the same core entities for construction and initialization, namely, the variables and constraints of the problem.

The second critical component that makes STAMPEDE easily modifiable is its command line argument parsing. The TCLAP library is a small, header only library that provides a simple mechanism for parsing command line arguments. It contains a number of classes used to define various kinds of argument types typically encountered on the command line, such as a `SwitchArg` (e.g., the common `-h` flag which takes no arguments), or a `ValueArg` (e.g., `–num-retries 10`). Users can build up a set of expected, required, or mutually exclusive arguments, pass in an array of strings corresponding to the command line arguments, and TCLAP will verify the arguments

[2]The `configurator` argument is an element of STAMPEDE that will be covered below in Section 3.2.3.

meet the specified requirements and provide a set of objects containing the parsed values.

In addition to the existing argument types defined in TCLAP, a new type, the `SubValueArg` was defined and added to an internal fork of the library. A `SubValueArg` is just a value argument, but with the addition of an arbitrary number of additional arguments contained inside of braces following the value. For example, a propagator might require some configuration, such as a switch to enable an experimental feature and an integer representing a timeout (e.g., `./Stampede -s BasicSearch {–consistencyPropagator MyNewProp {–timeout 10 –enableExperiment} –varOrder DomDeg}`).

The flags inside the braces are a completely independent set of command line arguments that can be verified and checked according to the propagator's needs. In fact, *any* object registered with the factory has its own set of command line arguments accompanied by an automatically generated help message. For example, passing `–help` to the GAC2001 propagator in STAMPEDE produces the help message displayed in Figure 3.5. While more recent C++ argument parsing libraries have made this feature more of a de facto standard, at the time of the implementation of STAMPEDE (circa 2014) it was fairly novel.

Notably, this feature allows STAMPEDE to arbitrarily nest propagators inside of other propagators without altering any code, assuming the propagator allows taking another propagator as an argument. This was used, in part, to create a new type of propagator (a *driver*) to selectively enforce varying levels of consistency to subproblems of the CSP derived from the properties of the problem itself [Woodward *et al.*, 2018; Woodward, 2018]. But most importantly, it makes STAMPEDE and all its individual components highly and easily configurable.

```
   GAC2001  -q <FIFO|lex|weighted> [--] [--version] [--serverHelp] [-h]


Where:

   -q <FIFO|lex|weighted>,  --queueType <FIFO|lex|weighted>
     (required)  The queue to use
        FIFO: A first-in, first-out queue
        lex: A lexicographical ordering of Table Constraints
        weighted: A priority queue that gives priority to the element with the highest constraint weight

   --,  --ignore_rest
     Ignores the rest of the labeled arguments following this flag.

   --version
     Displays version information and exits.

   --serverHelp
     Displays usage information but does not exit.

   -h,  --help
     Displays usage information and exits.


   A gac consistency algorithm
```

Figure 3.5: Example output of a help message for an object in STAMPEDE.

3.2.3 Consistency Propagators and Configuration

Consistency propagators are the major differentiating element of STAMPEDE from its contemporary solvers. Other solvers typically (ubiquitously, to our knowledge) enforce consistency properties at the granularity of a single constraint. That is, each propagator ensures that a given *constraint* adheres to the consistency property enforced by the propagator. Propagators in STAMPEDE, however, are responsible for ensuring that an entire *collection of constraints* adhere to a consistency property.

Of course, other solvers must have some mechanism for propagating changes caused by the enforcement of a consistency property to other variables and constraints, but those tend to be treated as implementation details, hidden from end users, with assumptions around a one-to-one mapping of constraint to propagator. CHOCO, for example, states

> "The Propagate component is less prone to be modified, it will not be described here. However, its interface is minimalist and can be easily

implemented." [Prud'homme, 2022a]

Perhaps it would not necessarily be *impossible* to implement a relational consistency propagator in another solver, but doing so would bend the assumptions these solvers were designed around, and likely produce very complex, error-prone code (were it even possible to do so without altering the CSP model itself).

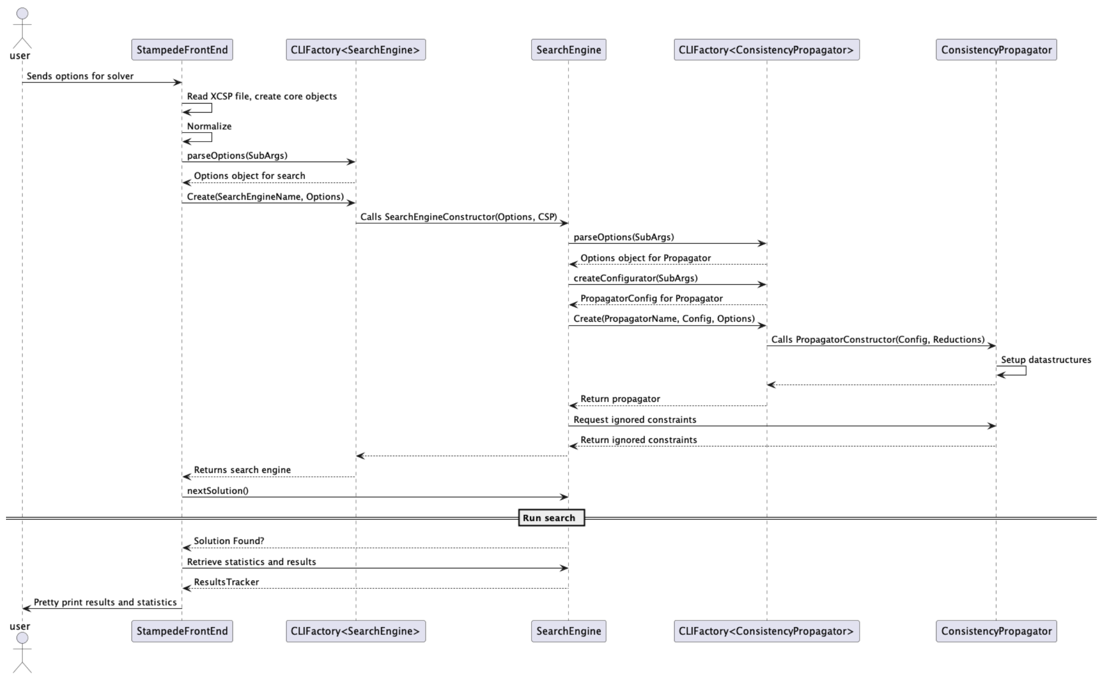

user
StampedeFrontEnd
CLIFactory<SearchEngine>
SearchEngine
CLIFactory<ConsistencyPropagator>
ConsistencyPropagator
Sends options for solver
Read XCSP file, create core objects
Normalize
parseOptions(SubArgs)
Options object for search
Create(SearchEngineName, Options)
Calls SearchEngineConstructor(Options, CSP)
parseOptions(SubArgs)
Options object for Propagator
createConfigurator(SubArgs)
PropagatorConfig for Propagator
Create(PropagatorName, Config, Options)
Calls PropagatorConstructor(Config, Reductions)
Setup datastructures
Return propagator
Request ignored constraints
Return ignored constraints
Returns search engine
nextSolution()
Run search
Solution Found?
Retrieve statistics and results
ResultsTracker
Pretty print results and statistics

Figure 3.6: Sequence diagram of initialization of Stampede.

Creating a new propagator and using it in search in Stampede is a fairly straightforward process for users wishing to add new propagators or search engines, but belies a fair amount of complexity under the surface. Figure 3.6 shows the initialization sequence typical for a simple consistency propagator. Users wishing to write new propagators only need be concerned with the final steps of the initialization, whereupon construction the propagator shall create all necessary data structures that it will use during search (graphs, book keeping, etc.).

Writing a new propagator is frequently contained to a single header and implementation file, with dependencies only arising for common support classes like graph representations. A helper macro such as the one in figure 3.7 is placed at the top of the propagator's implementation file, which creates the required registry entries in the `CLIFactory`. That is all that is required to begin using the propagator inside of any implemented search engine in STAMPEDE.

```
REGISTER_PROPAGATOR("GAC2001",
                    "Enforces GAC2001 on the table "
                    "constraints in the problem",
                    GAC2001,
                    GACHelpers::GAC2001Options,
                    GACHelpers::GAC2001Config)
```

Figure 3.7: Helper macro to enable GAC2001 propagator use in STAMPEDE.

The registration macro requires two classes aside from the propagator itself. The first is a class which contains the TCLAP logic used to parse and define the valid options for the propagator. This class must be a child of the `ArgumentInterface` class (e.g., `GACHelpers::GAC2001Options` referenced in Figure 3.7). `ArgumentInterface` contains only a single abstract function, `clone()`, which is used to make a deep copy of any objects used by the options class. The second class is a child of the `PropagatorConfigurator`[3] interface.

Figure 3.8 shows the relationships between the `PropagatorConfigurator` interface, two helper classes used to simplify the creation of concrete instances of the interface, and two such concrete implementations. The `Visitable` [4] class referenced in Figure 3.8 is an alias for a `std::variant`, a C++ class that is frequently used to implement visitor patterns [Gamma *et al.*, 1994], and is a type-safe union of classes. In the case of the `Visitable` class, the union is comprised of all concrete constraint and assignable

[3]An admittedly terrible name

[4]Another admittedly terrible name

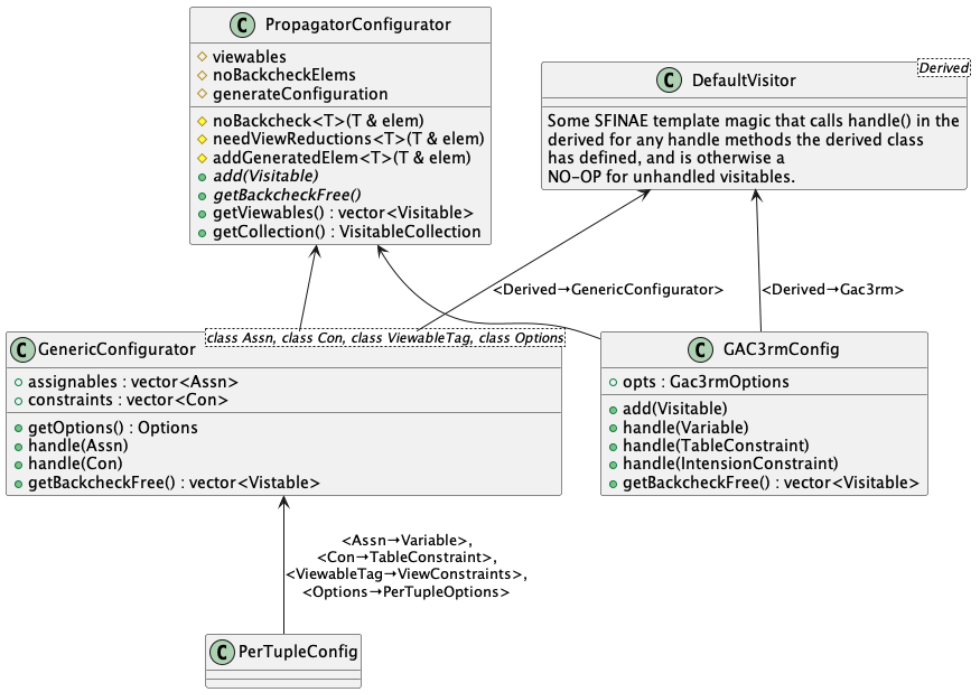

PropagatorConfigurator
viewables
noBackcheckElems
generateConfiguration
noBackcheck<T>(T & elem)
needViewReductions<T>(T & elem)
addGeneratedElem<T>(T & elem)
add(Visitable)
getBackcheckFree()
getViewables() : vector<Visitable>
getCollection() : VisitableCollection
DefaultVisitor
Derived
Some SFINAE template magic that calls handle() in the derived for any handle methods the derived class has defined, and is otherwise a NO-OP for unhandled visitables.
<Derived→GenericConfigurator>
<Derived→Gac3rm>
GenericConfigurator
class Assn, class Con, class ViewableTag, class Options
assignables : vector<Assn>
constraints : vector<Con>
getOptions() : Options
handle(Assn)
handle(Con)
getBackcheckFree() : vector<Vistable>
GAC3rmConfig
opts : Gac3rmOptions
add(Visitable)
handle(Variable)
handle(TableConstraint)
handle(IntensionConstraint)
getBackcheckFree() : vector<Visitable>
<Assn→Variable>,
<Con→TableConstraint>,
<ViewableTag→ViewConstraints>,
<Options→PerTupleOptions>
PerTupleConfig

Figure 3.8: Diagram of various classes used to configure propagators.

classes. A concrete implementation of a PropagatorConfigurator is given a set of Visitable objects. The PropagatorConfigurator must then examine each and decide which are relevant to its operation (and which are not) by *visiting* each Visitable in the collection.

The DefaultVisitor is a helper class that uses CRTP to allow specific PropagatorConfigurator implementations to handle the specific constraints and assignables that are relevant to them without needing to explicitly define every handle method for the classes defined within Visitable. In fact, the PropagatorConfigurator allows developers to craft propagators that can operate on very specific constraints (e.g., only constraints of a specific arity, with a certain name, or a particular connectivity property within the CSP).

The GenericConfigurator class in Figure 3.8 handles the most common use-case for propagators, where the propagator enforces a consistency over a specific kind of assignable (e.g., variables) and a specific kind of constraint (e.g., table constraints). Most propagator classes can use this by inheriting it with their chosen assignable and constraint types as template parameters. The `ViewableTag` template parameter is used to differentiate between the specific kinds of changes to the problem state that the propagator needs to be notified about: changes to the assignables, constraints, both, or neither.

These machinations serve two purposes. The first is to communicate information back to the search engine prior to search begins about any constraints in the problem that the propagator (or set of propagators) being used do not handle. These constraints will need to use back-checking to ensure consistency. The second is to ensure that after the initial setup, the propagator implementation will only ever see variables and constraints that it expects, and reduces the amount of time required to insert new implementations into STAMPEDE.

Table 3.1 shows the consistency algorithms currently available in STAMPEDE at the time of writing. The breadth of the available algorithms and relatively low amount of time required to implement these algorithms in STAMPEDE supports our claims of its flexibility and ease-of-use, and serves to validate the overall architecture of its design.

To summarize, the combination of the classes outlined in Section 3.2.1 and the `PropagatorConfigurator` outlined in this section makes it fairly trivial to begin developing a new propagator in STAMPEDE: Adding new command line arguments for a propagator requires modifying *only* the corresponding options class for the propagator, and adding an entirely new propagator requires *only* setting up the configuration class (which can be as simple as instantiating a `GenericConfigurator`) and using a macro.

Table 3.1: Available consistency algorithms in Stampede.

Algorithm	Consistency Property	Notes
AllSol	m-wise	
AllSolFB	m-wise	Option of AllSol
Backcheck	-	
CompactTable	GAC	
FC3rm	Forward Checking	Uses GAC3rm
GAC2001	GAC	
GAC3rm	GAC3rm	
LivingSTR	GAC	
MaxRPWC2	MaxRPWC	
NIC	NIC	
PC2001	PC2001	Binary only
POAC1	POAC1	
POACPartial	Partial POAC	
POACQ	POACQ	
PW-AC1	PWC	
PW-AC2	PWC	
PW-CT	fPWC	
PerTuple	m-wise	
PerFB	m-wise	Option of PerTuple
RNIC	RNIC	
RNICSingleQueue	RNIC	
SAC	SAC w/POAC-1	
SACQ	SAC w/POAC-Q	
STR1	GAC	
STR2	GAC	
STR3	GAC	
STRBit	GAC	
TriangleH3C	Triangle H3C	
TriangleH3CBit	Triangle H3C	
TrianglePPC	Triangle PPC	
TrianglePPCBit	Triangle PPC	
eSTR1	fPWC	Supports min dual and weakened
eSTR2	fPWC	Supports min dual and weakened
sCDC1	sCDC1	

3.3 Leveraging Modularity to Give Rise to Novel Relational Consistencies

One of the fundamental ideas that served as a north star when designing Stampede was ensuring components could be swapped in and out of algorithms without consistently requiring edge case handling or glue to bind disparate pieces together. The final implementation reflects this goal in a few (somewhat unexpected) ways due to the emergent behavior that arose from the combination of shifting propagators to enforce a consistency to the problem state (as mentioned in Section 3.2.3), treating relations as fundamentally changeable and first-class objects, and the ability to mix-and-match components.

3.3.1 Algorithms for Enforcing Hyper-3 Consistency

The decision to have `TableConstraint` objects also act as `Assignable`s mentioned in Section 3.2.1 complicated portions of Stampede, but gave rise to some unexpected and powerful emergent behavior. One notable example are the family of algorithms used to enforce Hyper-3 Consistency and its weaker variants [Woodward, 2018].

Hyper-3 Consistency is a consistency property that ensures that every two relations can be consistently extended to a third [Jégou, 1993]. This property mirrors the more familiar Path Consistency (PC), which ensure that every two *variables* can be consistently extended to a third. There also exist weakened versions of PC, Conservative Path Consistency (CPC) [Debruyne, 1999] and Partial Path Consistency (PPC) [Bliek and Sam-Haroud, 1999]. Each of these algorithms requires removing combinations of values from the constraints incident to the variables. Prior to the work presented by Woodward [2018] that was developed using Stampede, no algorithm for enforcing the many variants of Hyper-3 Consistency existed, as doing so would require operating on the dual encoding, triangulating it, and marking combinations of *tuples* as no-goods.

However, algorithms for enforcing Hyper-3 Consistency and its variants arose naturally from the relationships between table constraints, equality constraints, and the `Assignable` in STAMPEDE. In fact, the algorithms for enforcing the PC family of consistencies were carried over to the Hyper-3 Consistency family of consistencies almost directly, due to the explicit representation of equality constraints and the inheritance hierarchy that defines a table constraint as an assignable entity. Triangulating the dual graph for Hyper-3 Consistency algorithms used the same code that was used to triangulate the primal graph for the path consistency algorithms, and the enumeration of tuples within an equality constraint was accomplished by creating new table constraints for each equality constraints, where the "variables" in the scope of the generated table constraints were the original table constraints.

3.3.2 Drivers

Search engines in STAMPEDE are completely agnostic to the propagator and level of consistency they run, so long as the propagator accurately reports the constraints it will ensure at least the minimum level of consistency required to prevent backchecking. When execution is handed off from search to a propagator, the propagator is free to choose what consistency to enforce, where to enforce it, and how often to enforce it. This flexibility led to the eventual evolution of two distinct categories of `ConsistencyPropagator`s.

The first category hews closely to the traditional concept of a propagator. Generally, these operate on a specific kind of constraint, adding elements that need to be revised to a queue until quiescence is reached (i.e., the remaining problem adheres to the consistency property enforced by the propagator). The propagators for CompactTable and GAC2001, for example, fall into this category.

The second category is what we would come to refer to as *drivers*. A driver in Stampede is just a child of the `ConsistencyPropagator` class, but enforces some form of hybrid consistency, employing one or more other propagators within it. Configuring a driver is the same as any other propagator, though drivers are responsible for also accurately reporting (to the search engine) what constraints are handled by all propagators that the driver itself creates.

The tree decomposition cluster-driven consistency, R($*$,$|\psi|$)C [Karakashian *et al.*, 2011, 2013] is implemented using one such driver. The propagator, `ClusterBased-Driver`, first constructs the tree decomposition. Then, for each cluster in the tree decomposition, it creates a separate propagator. In the case where we wish to enforce R($*$,$|\psi|$)C, each propagator is one of PerTuple or AllSol [Karakashian *et al.*, 2010a, 2013]. The original implementations of the algorithms used to enforce R($*$,$|\psi|$)C were done in scsp, but fit naturally within the framework Stampede provides. Other selective propagators defined in the literature also fit naturally into this model. For example, `MABDriver` uses a multi-armed bandit to probabilistically enforce one of an arbitrary number of consistencies [Balafrej *et al.*, 2015]. Table 3.2 shows the list of all currently implemented drivers and a brief description of their purpose.

Importantly, any alterations to the CSP model are isolated to the class which altered the model. For example, constraints added by a tree decomposition to bolster propagation between cluster are not relayed to search or other propagators, *unless specifically requested* by the user running the search. The only entity that needs to be notified of the additional constraints would be the `Reductions` class in order to

Table 3.2: List of drivers that STAMPEDE supports.

Driver	Description
ClusterBasedDriver	Enforces a consistency on each cluster of a tree decomposition of the CSP
DensityDriver	Enforces a consistency if primal graph density is below a threshold
MABDriver	Uses multi-armed bandit to selectively enforce other consistency algorithms
PortfolioDriver	Enforces consistency from a given portiolio
SyncDriver	Used to project/select changes to/from table constraints
TriggerDriver	Enforces a list of arbitrary consistency algorithms
RepeatDriver	Repeats enforcement of a set of consistencies

notify any propagators that *do* share the generated constraints of removals.

The ability to compose drivers from other propagators (or even other drivers) also gave rise to several novel types of consistency properties. The work presented by Woodward [2018] and Woodward *et al.* [2018] is one such example. `DensityDriver` runs a consistency (chosen at run-time) only if the primal graph's density is below some specified threshold. The `TriggerDriver` was used to implement methods that control when, where, and how much of an HLC to enforce at specific search nodes based on so-called "triggers" in the problem, and was empirically shown to improve search performance. The work from Geschwender *et al.* [2016] and Geschwender [2018] uses a driver to select one of several propagators at each node in search from a portfolio based on a statistical classifier chosen at run-time.

3.4 Research Oriented

Ensuring STAMPEDE was suitable for conducting research on CSPs was of paramount import in its design. This section outlines a few of the ways STAMPEDE meets the demands of pursuing new directions of research, with specific regards to high-level and relational consistencies.

3.4.1 Ordering and Reproducibility

A major component of research involves A/B comparisons between different algorithms (or modifications to existing algorithms). Ensuring reproducibility when running the same CSP instance requires deterministic ordering of every component of the CSP. Stampede solves this in part through the ID mechanism described in Section 3.2.1. Each core element of the CSP has at least two unique IDs (one universal amongst all core elements, and another unique to its class), and the IDs are created in the order that the entities are read in from the XML file that describes a CSP. These IDs are thus safe to use for deterministic ordering as the XML files are static, and cheaper than performing lexicographical comparisons of the strings corresponding to the names of the objects. All containers built for Stampede (such as sparse sets and intrusive containers) use these IDs for lookup and ordering.

Another possible source of randomness are the variable and value ordering heuristics used to select the next variable-value pair to instantiate during search. Stampede offers a number of heuristics to use that will provide a static ordering with respect to a given propagation algorithm. However, ordering heuristics such as $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ may begin to explore radically different search trees depending on which constraints are reported as the cause of a domain wipeout. While this can be mitigated to some extent by ordering propagation queues, relational consistencies in particular are very susceptible to changes induced by $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ as they may find domain wipeouts in entirely different portions of the CSP than something enforcing GAC would.

Stampede also limits the number of design patterns that can contribute to surprising execution order like signals, observers, and raw callbacks. These are present in some situations where not having them present would cause a significant amount of burden (such as constraint weight incrementing in ordering heuristics), but in situations where they are used, they do not affect search state. This ensures simply reading the code for Stampede can give an accurate impression on execution order.

3.4.2 Interacting with Stampede

In addition to the more conventional ways that Stampede can aid with research, it also provides a web-browser based graphical user interface (GUI). The GUI is effectively broken into two pieces. The classes shown in Figure 3.9 are the key classes that form the backend of the GUI. If Stampede is started as a daemon (rather than used to solve a single instance), it creates an internal websocket server. Individual classes in Stampede can inherit from the `GuiProducer` class to send JSON-formatted messages over the websocket. The only required component of the JSON message is a UUID that corresponds to the object in Stampede that is responsible for sending and receiving messages to and from the frontend.

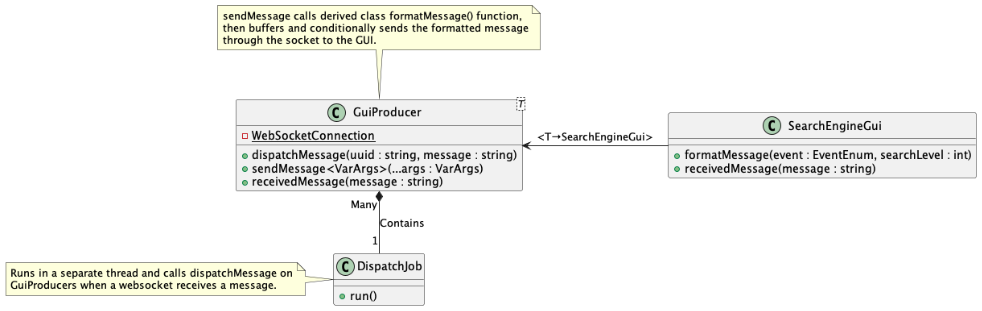

sendMessage calls derived class formatMessage() function, then buffers and conditionally sends the formatted message through the socket to the GUI.
GuiProducer
T
WebSocketConnection
dispatchMessage(uuid : string, message : string)
sendMessage<VarArgs>(...args : VarArgs)
receivedMessage(message : string)
<T→SearchEngineGui>
SearchEngineGui
formatMessage(event : EventEnum, searchLevel : int)
receivedMessage(message : string)
Many
Contains
1
DispatchJob
run()
Runs in a separate thread and calls dispatchMessage on GuiProducers when a websocket receives a message.

Figure 3.9: A simplified UML diagram Stampede's websocket GUI architecture.

The bulk of the work is handled behind a few layers of abstractions; users wishing to add a visual element to any of their classes just define the information required to visualize their class. For example, a user wishing to add a visualization to show the graph of a CSP would only need to send the variables and constraints in the JSON format of their choosing (or the nodes and edges of one of the graph formulations).

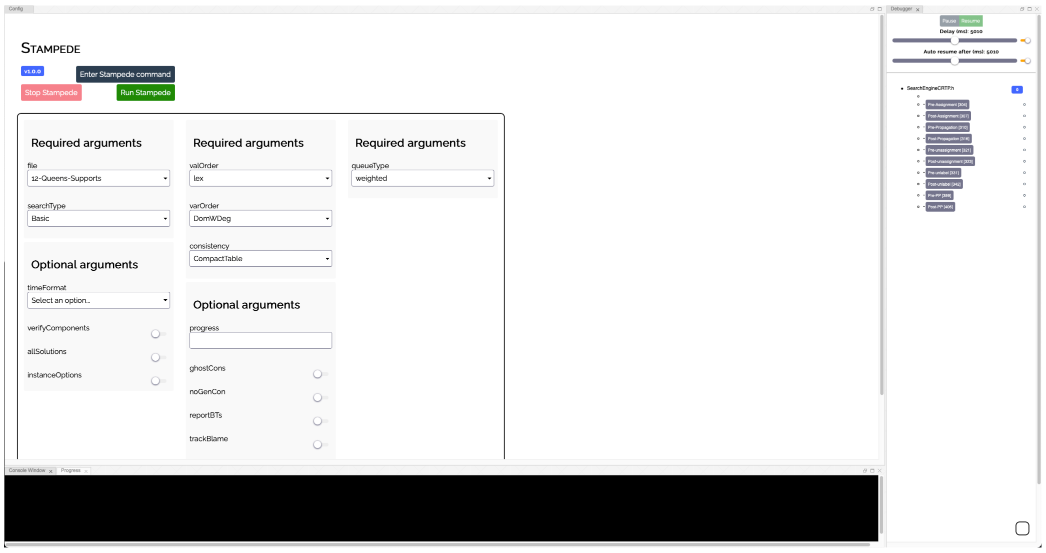
Users can also interact with STAMPEDE from the GUI. Messages sent from the GUI are received by a `DispatchJob` class, which polls to check for new messages in a separate thread from the one STAMPEDE is using to solve the instance, and forwards the message to its intended target based on the UUID in the message. Classes that inherit from `GuiProducer` are automatically registered with the dispatch job in a similar manner to how classes are registered with the `CLIFactory`. Messages are likewise dispatched *to* the GUI on a separate thread to minimize the overhead of running STAMPEDE in this mode.

There were two primary motivators for setting up visualizations for STAMPEDE with this architecture. The first is the vast array of data visualization libraries that are available for Javascript and other browser-based languages. The second is the ease of implementing new UI elements in a web page relative to a C++ application which was not initially designed to have a GUI at all. A service-based implementation allows for ad hoc additions of new GUI representations for arbitrary classes without significant time needing to be spent dealing with C++-specific frameworks (which have a tendency to be cumbersome and intrusive in this author's humble opinion).

While this architecture technically allows for any software capable of sending and receiving JSON messages over a websocket to act as a frontend for STAMPEDE, the current implementation uses a web page to interact with the solver[5]. Figure 3.10 shows two screenshots of the frontend.

Figure 3.10a shows the landing page, which consists of all of the available options the solver provides, dynamically loaded on mouse over. The arguments in the frontend mirror the TCLAP command line arguments exactly without any additional effort from developers. When STAMPEDE is launched as a daemon, a new handler class for TCLAP

[5]An undergraduate research assistant, Denis Komissarov, took on the bulk of front-end implementation under guidance from the author.



STAMPEDE

v1.0.0

Enter Stampede command

Stop Stampede

Run Stampede

Required arguments

file: 12-Queens-Supports

searchType: Basic

Optional arguments

timeFormat: Select an option...

verifyComponents

allSolutions

instanceOptions

Required arguments

valOrder: lex

varOrder: DomWDeg

consistency: CompactTable

Optional arguments

progress

ghostCons

noGenCon

reportBTs

trackBlame

Required arguments

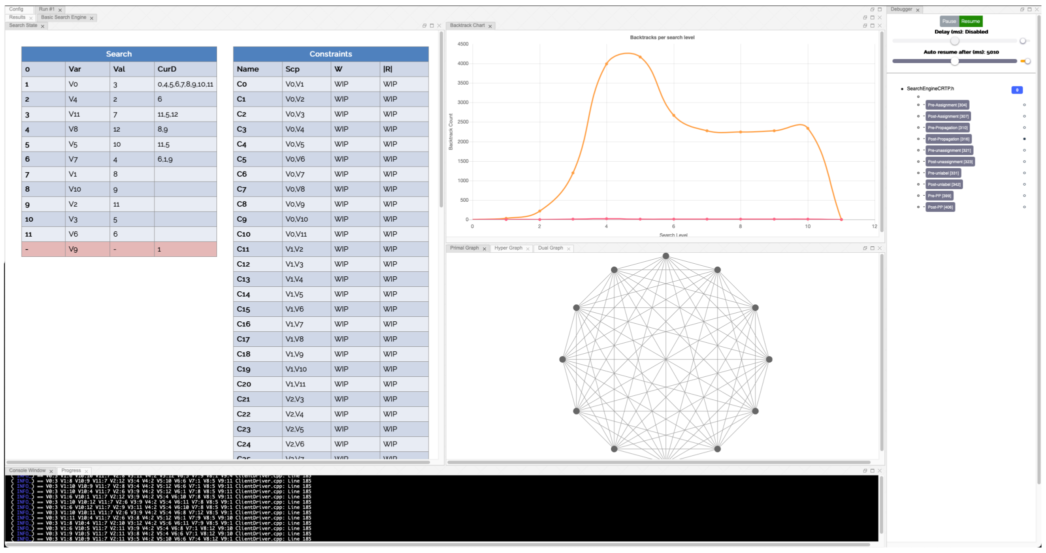
queueType: weighted

Delay (ms): 5010

Auto resume after (ms): 5010

- SearchEngineCRTP.h 0
  - Pre-Assignment [304]
  - Post-Assignment [307]
  - Pre-Propagation [310]
  - Post-Propagation [316]
  - Pre-unassignment [321]
  - Post-unassignment [323]
  - Pre-unlabel [331]
  - Post-unlabel [342]
  - Pre-PP [399]
  - Post-PP [406]

(a) During setup.



Search			
0	Var	Val	CurD
1	V0	3	0,4,5,6,7,8,9,10,11
2	V4	2	6
3	V11	7	11,5,12
4	V8	12	8,9
5	V5	10	11,5
6	V7	4	6,1,9
7	V1	8	
8	V10	9	
9	V2	11	
10	V3	5	
11	V6	6	
-	V9	-	1

Constraints			
Name	Scp	W	|R|
C0	V0,V1	WIP	WIP
C1	V0,V2	WIP	WIP
C2	V0,V3	WIP	WIP
C3	V0,V4	WIP	WIP
C4	V0,V5	WIP	WIP
C5	V0,V6	WIP	WIP
C6	V0,V7	WIP	WIP
C7	V0,V8	WIP	WIP
C8	V0,V9	WIP	WIP
C9	V0,V10	WIP	WIP
C10	V0,V11	WIP	WIP
C11	V1,V2	WIP	WIP
C12	V1,V3	WIP	WIP
C13	V1,V4	WIP	WIP
C14	V1,V5	WIP	WIP
C15	V1,V6	WIP	WIP
C16	V1,V7	WIP	WIP
C17	V1,V8	WIP	WIP
C18	V1,V9	WIP	WIP
C19	V1,V10	WIP	WIP
C20	V1,V11	WIP	WIP
C21	V2,V3	WIP	WIP
C22	V2,V4	WIP	WIP
C23	V2,V5	WIP	WIP
C24	V2,V6	WIP	WIP

```
( INFO_) == V0:3 V1:8 V10:9 V11:7 V2:12 V3:4 V4:2 V5:10 V6:6 V7:1 V8:5 V9:11 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:10 V10:9 V11:7 V2:8 V3:4 V4:2 V5:12 V6:6 V7:1 V8:5 V9:11 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:10 V10:4 V11:7 V2:6 V3:9 V4:2 V5:12 V6:1 V7:8 V8:5 V9:11 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:6 V10:1 V11:7 V2:12 V3:9 V4:2 V5:4 V6:10 V7:8 V8:5 V9:11 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:10 V10:12 V11:7 V2:6 V3:9 V4:2 V5:4 V6:11 V7:8 V8:5 V9:1 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:6 V10:12 V11:7 V2:9 V3:11 V4:2 V5:4 V6:10 V7:8 V8:5 V9:1 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:10 V10:11 V11:7 V2:6 V3:9 V4:2 V5:4 V6:8 V7:12 V8:5 V9:1 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:11 V10:4 V11:7 V2:6 V3:8 V4:2 V5:12 V6:1 V7:9 V8:5 V9:10 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:8 V10:4 V11:7 V2:10 V3:12 V4:2 V5:6 V6:11 V7:9 V8:5 V9:1 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:6 V10:5 V11:7 V2:11 V3:9 V4:2 V5:4 V6:8 V7:1 V8:12 V9:10 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:9 V10:5 V11:7 V2:11 V3:8 V4:2 V5:4 V6:6 V7:1 V8:12 V9:10 ClientDriver.cpp: Line 185
( INFO_) == V0:3 V1:8 V10:9 V11:7 V2:11 V3:5 V4:2 V5:10 V6:6 V7:4 V8:12 V9:1 ClientDriver.cpp: Line 185
```

Delay (ms): Disabled

Auto resume after (ms): 5010

- SearchEngineCRTP.h 0
  - Pre-Assignment [304]
  - Post-Assignment [307]
  - Pre-Propagation [310]
  - Post-Propagation [316]
  - Pre-unassignment [321]
  - Post-unassignment [323]
  - Pre-unlabel [331]
  - Post-unlabel [342]
  - Pre-PP [399]
  - Post-PP [406]

(b) During search.

Figure 3.10: Images showcasing STAMPEDE's browser-based GUI and debugger.

is created instead of its default handler, that inherits from the `GuiProducer` class and communicates all of the available options for a given `SubArg` to the GUI by formatting a portion of the help message (which describes all available options for a `SubArg`) and forwarding it to the frontend. STAMPEDE's logs and output are forwarded to the GUI using a similar mechanism, and displayed in the console window at the bottom of the page.

Figure 3.10b shows the GUI for STAMPEDE while search is running. Of particular note is the column on the right side of the figure. These buttons allow users to interact with STAMPEDE *as search is running* to dynamically pause the search and examine the state of the problem, as well as enable breakpoints (created during STAMPEDE's compilation) where execution will pause until manually resumed. It also contains a feature to automatically resume search after being paused for a given duration to slow search down enough to allow human users to examine search as it proceeds. The breakpoints are populated as part of the build, and can be placed at any location in STAMPEDE. Frequently, debugging new algorithms and ideas requires extremely granular insight into the CSP state and datastructures being used in (for example) a propagator. This feature allows researchers to step through search to help identify errors in their code.

This GUI was used to form new visualizations used to help validate and understand the performance of propagators: the number of backtracks per depth (BpD) and the number of consistency calls per depth (CpD) [Howell *et al.*, 2020; Woodward *et al.*, 2018; Woodward, 2018].

Summary

In this chapter we described Stampede, a CSP solver built specifically to aid in the development and research of relational consistency propagators.

While Stampede succeeds with respect to the goals laid out in Section 3.1.3, it is not without faults. Though not a goal, the overall performance could likely be substantially improved by redesigning its core data structures with cache locality and dense representations in mind (a focus early in the design was on the Big-O performance of datastructures and algorithms, rather than real-world performance). The GUI was produced without an explicit design in mind, as it was initially produced as a proof-of-concept rather than a full-fledged feature, but could be retooled to use a purpose-built RPC service (e.g., gRPC) instead of websockets and custom-formatted JSON. Beyond increasing its speed and reliability, a dedicated RPC service would provide a unified message form (e.g., protos) rather than the more ad-hoc JSON messages currently used. And though the inheritance structure of `Assignable`, `TableConstraint`, and `Variable` resulted in some beneficial emergent behavior, the same could likely have been accomplished through other, less complicated means.

Despite the above shortcomings (and many others not listed here), the merits of Stampede have already been validated in the context of

1. New reactive strategies for monitoring, controlling, and selectively triggering consistency [Woodward *et al.*, 2018; Woodward, 2018],

2. Examination of performance on different models of nonogram puzzles using HLC and GAC [Tran, 2019],

3. Controlling the application of minimality algorithms over subproblems [Geschwender *et al.*, 2016; Geschwender, 2018],

4. The development and validation of new higher-level consistency algorithms [Schneider and Choueiry, 2018; Woodward *et al.*, 2017], and

5. New approaches for visualizing the performance of propagators during search [Howell *et al.*, 2020; Woodward *et al.*, 2018; Woodward, 2018].

Thus, it is our belief that STAMPEDE is clearly a valuable tool for the research and development of novel constraint processing techniques and ideas.

Chapter 4

Pairwise Consistency Algorithms

GAC has long been the focus of extensive research in constraint processing, and for good reason: it lends itself towards simple yet highly effective algorithms. The low cost and effectiveness of GAC algorithms when paired with an ordering heuristic like $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ have made them the de facto baseline for research. The current state-of-the-art in GAC algorithms are CompactTable [Demeulenaere *et al.*, 2016] and STRbit [Wang *et al.*, 2016], both of which use bitsets to quickly check for supports and perform tabular reduction – the process of removing invalid tuples from constraints.

As with GAC, pairwise consistency is arguably the most straightforward relational consistency. Its definition is equivalent to arc consistency on the dual graph. A naive approach to enforcing PWC would be to enforce AC on the dual encoding of the CSP, but this would not be efficient as the number of tuples tends to be very large in most problems, and this approach would not exploit properties inherit to the dual encoding that can be used to more efficiently enforce PWC. Most existing work focuses on enforcing PWC in a variable/domain-centric manner (e.g., RPWC and eSTR), but few algorithms treat the relations themselves as first class entities. This is likely in large part due to the variable-based algorithms for enforcing PWC generally outperforming their relational consistency competitors.

In this chapter we provide a brief overview of one of the most recent variable based methods for enforcing PWC (eSTR), and an in-depth explanation of one of the only relational methods for enforcing PWC (PW-AC). We then provide two new algorithms. The first algorithm, PW-AC2, greatly improves on PW-AC, and we show that it can outperform even the variable based state-of-the-art algorithms for enforcing PWC. The second algorithm, PW-CT, uses lessons learned from constructing PW-AC2 and merges those with the current state-of-the-art GAC algorithm CompactTable, and was published in [Schneider and Choueiry, 2018]. We show that PW-CT is competitive with CompactTable and STRbit in many problems and can in fact best it in several. It's also capable of beating other GAC algorithms in nearly all cases, and clearly and definitively outperforms all other algorithms for enforcing PWC.

4.1 Current State-of-the-Art for enforcing PWC

The algorithm eSTR was introduced by Lecoutre *et al.* [2013] along with its variant eSTR2. They are based on the Simple Tabular Reduction (STR) family of algorithms which not only enforce GAC, but also remove tuples from relations that contain any non-GAC values in the domains of the variables in its scope. This *reduction* of tuples from the table constraints helps speed up subsequent searches for support for GAC values, effectively improving the performance of enforcing GAC.

At a high level, STR algorithms operate by ensuring that every value in the domain of a variable has a supporting tuple in each constraint by iterating over the tuples in the constraint. The algorithms store a pointer to the last known support found for a variable-value pair so that it can resume search from that position in the future (as once a tuple in a constraint is determined to not be a support, it will remain that way until a backtrack occurs). While seeking the next support for a value, it skips over previously removed tuples in the constraint.

There are three variants of STR: STR, STR2, and STR3 [Lecoutre *et al.*, 2012; Lecoutre, 2011]. The difference between these algorithms is not relevant to the remainder of this dissertation; the salient takeaway is that STR2 is typically the best performing variant.

The eSTR algorithms enforce fPWC by extending the STR algorithms. In eSTR, after finding a tuple in a constraint that supports a value, eSTR then ensures that the tuple is also pairwise consistent with all neighboring constraints. It does this by maintaining counts of the number of tuples that are pairwise consistent with a given tuple in a constraint, and updating those counts upon the deletion of a tuple. If the count for a support is zero, eSTR continues looking for another tuple that supports the value. Algorithm eSTR^w is a modification of eSTR and enforces a weakened version of fPWC by not re-queuing a constraint after a PW-support is lost. The differences between eSTR and eSTR2 are simply the STR algorithm used as its base. As such, we use eSTR2 and eSTR2^w in this chapter as they are typically the most performant of all variations.

Algorithms HOSTR and maxRPWC+r [Paparrizou and Stergiou, 2016] enforce consistency properties that are weaker than PWC and incomparable to each other. fHOSTR, a variant of HOSTR, enforces fPWC, but was found by its authors to be too expensive relative to its weakened version. Lecoutre [2011] show that HOSTR and maxRPWC+r outperform STR2 on certain benchmarks.

Some approaches for enforcing higher order consistencies apply GAC after reformulating the CSP with new constraints or variables. Algorithm DkWC [Mairy *et al.*, 2014] enforces k-wise consistency by adding new hybrid constraints to the problem. The Factor Encoding (FE) enforces fPWC by adding new variables to the problem, thereby increasing the arity of constraints [Likitvivatanavong *et al.*, 2014]. A decomposition of the FE lessens the imposed arity increases from FE while still enforcing fPWC [Likitvivatanavong *et al.*, 2015]. However, we ignore these approaches because they modify the constraint network as part of their operation, which is not only expensive, but also tends to result in much more complicated algorithms and reasoning.

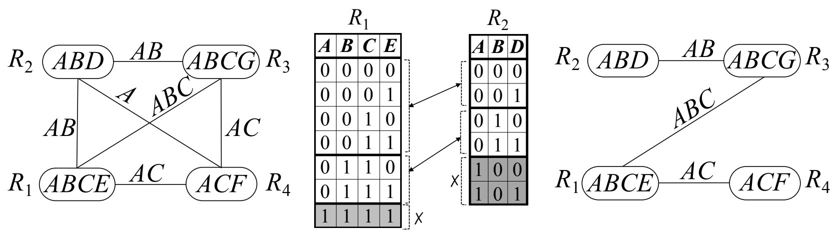


Figure 4.1: Dual graph (left), subscopes and blocks (center), a minimal dual graph (right)

The algorithm PW-AC was introduced by Samaras and Stergiou [2005]. This algorithm enforces PWC (*not* fPWC) by partitioning relations based on their piecewise functionality. In Figure 4.1, the subscope AB partitions each of the two relations R_1 and R_2 into three *blocks*. We define the *signature* of a block as the set of variable-value pairs of the inducing subscope (e.g., $\{\langle A, 0\rangle, \langle B, 0\rangle\}$). Thus, a signature is uniquely determined by a combination of a constraint, subscope, and tuple.

In PW-AC, for each pair of relations in the problem that have non-empty subscopes, the algorithm buckets tuples in each relation into blocks that share the same signature. Each block will have at most one pairwise support in its paired relation. If one of the blocks is empty, all tuples in the pairwise-consistent block of the other relation can be removed from the problem as all of those tuples have lost their support. These removals may cause other relations to lose their supports, propagating the deletions throughout the CSP. Enforcing fPWC after PWC using PW-AC requires projecting tuple deletions of the relations onto the variables in their scopes. This will remove any non-GAC values from the variables caused by the PWC deletions.

4.2 Techniques for Improving Pairwise Consistency

Below, we describe four distinct techniques to improve the performance of PWC algorithms. These methods can be combined or exploited in isolation.

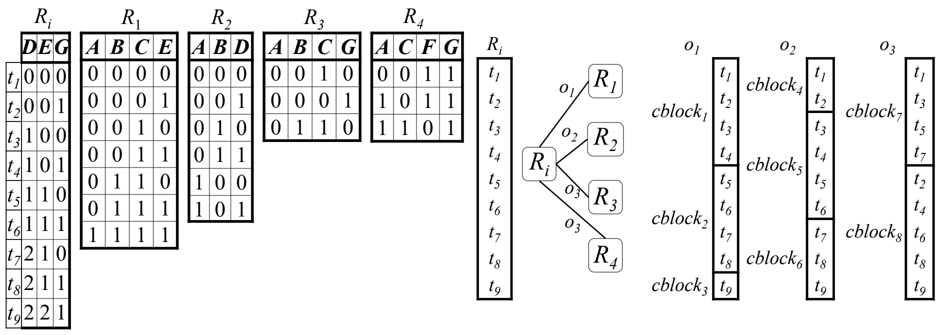
4.2.1 Piecewise Functionality

As mentioned in Chapter 2, Samaras and Stergiou [2005] exploit the piecewise-functional property of the equality constraints of the dual graph to infer the blocks of equivalent tuples of two constraints with shared variables. If a tuple τ in a constraint c_i does not have a PW-support in another constraint c_j, all tuples in the block induced by $\pi_{subscope(c_i,c_j)}(\tau)$ on c_i can be immediately removed. Further, all other *blocks* of tuples that are PW-supported by τ in all other neighboring constraints (i.e., blocks with the same signature) must also be deleted. This operation is in stark contrast with most GAC-based algorithms that search for supports one tuple at a time (with the notable exception of AC-5 [Hentenryck *et al.*, 1992]).

4.2.2 Refocusing Propagation on Subscopes

Algorithms for enforcing pairwise consistency usually operate on *every pair* of constraints with overlapping subscopes (e.g., PW-AC partitions relations pairwise, eSTR counts supports pairwise). Karakashian *et al.* [2010a] and Schneider *et al.* [2014] exploit the fact that, for a given subscope, all relations induce on another relation R_i the same unique partition. For example, in Figure 4.1, the blocks induced by subscope $\{A, B\}$ on relation R_1 are the same for *any* relation R_j such that $subscope(R_1, R_j) = \{A, B\}$.

Consider the more concrete example given in Figure 4.2, which shows a set of



R_i

	D	E	G
t_1	0	0	0
t_2	0	0	1
t_3	1	0	0
t_4	1	0	1
t_5	1	1	0
t_6	1	1	1
t_7	2	1	0
t_8	2	1	1
t_9	2	2	1

R_1

A	B	C	E
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	1	0
0	1	1	1
1	1	1	1

R_2

A	B	D
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

R_3

A	B	C	G
0	0	1	0
0	0	0	1
0	1	1	0

R_4

A	C	F	G
0	0	1	1
1	0	1	1
1	1	0	1

Figure 4.2: A set of relations and the partitions for relation R_i.

relations and a partitioning scheme for one of the relations. Each block is comprised of a disjoint set of tuples. Although there are four relations adjacent to R_i, we only partition R_i three times (once per unique subscope). Consequently, identifying and storing a relation's partitions based on unique subscopes rather than by the degree of a vertex in the dual graph can significantly reduce the memory requirements of algorithms that exploit the pairwise functionality of the equality constraints of the dual graph. Depending on the nature of the support checks in the algorithm, it may also reduce the number of required PWC checks.

4.2.3 Minimal Dual Graph

As stated in Section 2, we can remove redundant edges in the dual graph of a CSP without affecting the set of solutions. In fact, Janssen *et al.* [1989] show that enforcing PWC on a dual graph is equivalent to enforcing PWC on any of its minimal dual graphs. Importantly, removing redundant edges can reduce not only the degree of the graph (thus reducing the number of pairs of constraints over which a PWC algorithm must iterate) but also the number of unique subscopes that a PWC algorithm must take into consideration. For instance, in the example shown in Figure 4.1, removing redundant edges eliminates: (1) The need to compute and store the partitions of R_1

for the subscope $\{A, B\}$ and the partitions of R_3 for the subscope $\{A, C\}$ and (2) The subscope $\{A\}$ and the partition it induces on each of R_2 and R_4. Consequently, a minimal dual graph can reduce the number of neighbors of a constraint in the problem, the number of unique subscopes incident to a constraint, and may eliminate some subscopes from the problem entirely. We conclude that a PWC algorithm that operates on a minimal dual graph may reduce its memory requirements and increase its propagation speed because of the reduced number of subscopes to consider per constraint and the total number of unique subscopes.

4.2.4 Determining when GAC is enough to enforce PWC

In some situations, GAC is enough to enforce PWC between constraints. The algorithm eSTR, for example, only checks for PW-supports over "non-trivial" subscopes, which are subscopes with a cardinality strictly greater than one [Lecoutre *et al.*, 2013]. In fact, the particularity of constraints intersecting on at most one variable is discussed by Bessière *et al.* [2008] but PWC is inexplicably excluded from the corresponding theorem. Below, we restate this property and give a proof:

Proposition 1. *GAC is sufficient to enforce PWC over trivial subscopes.*

Proof. Consider the CSP $\mathcal{P}= (\mathcal{X}, \mathcal{D}, \mathcal{C})$. If a subscope is trivial (e.g., *subscope* $= \{x\} \subset \mathcal{X}$) the signature of each block induced by this subscope is one variable-value pair (e.g., $\langle x, a\rangle$). Thus, the block loses all PW-supports only if $\langle x, a\rangle$ is removed from the problem. If $\langle x, a\rangle$ is deleted, a tabular-reduction algorithm necessarily removes all tuples with $\langle x, a\rangle$ from the problem. On the other hand, if $\langle x, a\rangle$ is alive after enforcing GAC, then, by definition, $\forall c_i \in \mathcal{C}$ such that $x \in \text{scope}(c_i)$, there is at least one living tuple τin the relation of c_i such that $\pi_x(\tau) = a$. $\square$

We describe a particular situation, which arises during search, in which the above property holds even for non-trivial subscopes as long as GAC is enforced on a constraint prior to running a PWC algorithm:

Proposition 2. *GAC is sufficient to enforce PWC on a block induced by a non-trivial subscope whose signature includes a deleted variable-value pair.*

Proof. This proposition follows from Proposition 1. Consider a block b_i induced by a non-trivial subscope σ_i on the constraint c_i. If a dead variable-value pair $\langle x, a \rangle$ is in the block's signature, a tabular-reduction GAC algorithm removes all tuples with $\langle x, a \rangle$ from the problem, and as a result, it removes all the PW-supports of b_i from the relations of neighboring constraints because they necessarily also contain $\langle x, a \rangle$ in their signatures. □

Algorithm eSTR implicitly applies this principle by ensuring that all the variable-value pairs of a tuple are alive before checking whether or not the tuple has PW-supports in neighboring constraints. We exploit Proposition 2 in a PWC algorithm in a slightly more efficient manner, which is described in Section 4.5. Briefly: assume a CSP is already PWC. After a variable is instantiated, we run an STR-based GAC, which may delete tuples from constraints. We now need to process these deleted tuples because some of them may have been the last remaining tuples in blocks that were the PW-support of blocks in other constraints. In the case that a variable-value pair deleted by GAC appears in the signature of a block in which one of these deleted tuples appears, we can safely skip the processing we intended to do to enforce PWC because its result is ensured by GAC.

4.3 Integrating Improvements into existing PWC Algorithms

Below we introduce a first attempt at applying the improvements introduced in Section 4.2 by applying a subset of the suggested improvements to the algorithm PW-AC. This algorithm, denoted PW-AC2, is a direct successor of the PW-AC algorithm; the flow of the algorithm and manner of propagation is nearly identical to PW-AC. It also takes advantage of the piecewise functionality of table constraints in the dual graph as PW-AC does by partitioning relations into blocks. However, unlike PW-AC, which partitions every pair of relations connected in the dual graph, PW-AC2 partitions according to its relations' incident subscopes. PW-AC2 also uses the minimal dual graph to potentially reduce the number of subscopes required to partition against in the problem.

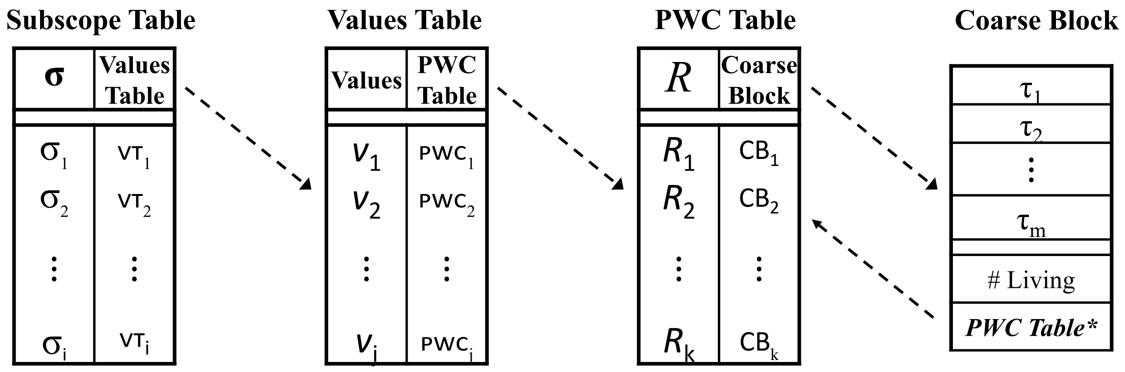


Figure 4.3: Data structures used to partition relations according to subscopes for PW-AC2.

The data structures shown in Figure 4.3 provide access to the blocks computed during preprocessing. The blocks are organized by a particular block's signature (e.g., its subscope, subtuple, and relation). These can be stored in a 3-dimensional table using a combination of hashmaps, sparse maps, and vectors, but for the purposes of explaining the algorithm we will refer to the structure as a simple array structure coarseBlocks[σ_i][v_i][R_i] where σ_i is a subscope in the problem, v_i is a subtuple (stored as a vector of domain values) for the variables in the subscope, and R_i is a relation incident to σ_i. If a combination or subset of the combination of σ_i, v_i, and R_i is not present in the table, it returns ***nil*** (e.g., if a subtuple v_i for subscope σ_i has not yet been recorded, coarseBlocks[σ_i][v_i] will return ***nil***, as will coarseBlocks[σ_i][v_i][R_i].

Algorithm 1 presents the required initialization steps for PW-AC2. The method `initStructs`, defined on line 1 of Algorithm 1 iterates over every edge in the dual graph. We assume that the dual graph g has already had its redundant edges removed, although this is not a hard requirement for the algorithm; the pseudocode presented here works regardless of whether the minimal dual graph is being used. However, we explicitly define PW-AC2 as running on the minimal dual graph.

Algorithm 1: Data Structure Initialization

```
1  Method initStructs($g$: A DualGraph of the CSP):
2      foreach edge $e \in g$ do
3          $R_i, R_j \leftarrow$ relations incident to $e$
4          subscope $\sigma_i \leftarrow$ scope($R_i$) $\cap$ scope($R_j$)
5          PartitionSubscope($R_i$, $\sigma_i$)
6          PartitionSubscope($R_j$, $\sigma_i$)
7  Method PartitionSubscope($R_i, \sigma_i$: A relation, subscope to partition):
8      if AlreadyPartitioned($R_i$,$\sigma_i$) then
9          return
10     foreach tuple $\tau \in R_i$ do
11         $subtuple_i \leftarrow \pi_{\sigma_i}(\tau)$
12         if coarseBlocks[$\sigma_i$][$subtuple_i$][$R_i$] $= nil$ then
13             coarseBlocks[$\sigma_i$][$subtuple_i$][$R_i$] $\leftarrow$new coarse block
14         tupleBlock $\leftarrow$ coarseBlocks[$\sigma_i$][$subtuple_i$][$R_i$]
15         tupleBlock.tuples $\leftarrow$ tupleBlock.tuples $\cup\ \tau$
16         tupleBlock.count $\leftarrow$ tupleBlock.count + 1
17         tupleBlock.relation $\leftarrow R_i$
18         tupleBlock.pwctable $\leftarrow$ coarseBlocks[$\sigma_i$][$subtuple_i$]
19         blockLookup[$R_i$][$\sigma_i$][$\tau$] $\leftarrow$ reference to tupleBlock
```

The method `PartitionSubscopes` defined on line 7 then checks if the given $\langle relation, subscope \rangle$ pair has already been partitioned. This can be implemented using any number of data structures and thrown away after the initialization is complete, so we omit the details of the implementation here for brevity. The algorithm then proceeds to iterate over each tuple of the relation. Lines 12–13 create a new coarse block if one for the current signature has not yet been created. The block is then modified to include a reference to the current tuple, a count denoting how many tuples are alive in the block, and a reference to the relation. An ancillary lookup table, `blockLookup`, is created to quickly find blocks that a tuple belongs to, provided a relation and a subscope.

Once the data structures are initialized by line 3 in Algorithm 2, a queue is populated with any blocks that are initially missing pairwise supports. The next call to `Propagate` will iterate over this queue. Method `Revise` takes a block b_i (corresponding to a relation R_i) that was popped from the queue and removes the tuples in b_i one at a time. The `blockLookup` table is used in `Revise` to lookup (in constant time) other blocks in R_i being revised that contain the removed tuple. Lines 24–28 decrement the count of living tuples for each b_j in R_i that contain the newly deleted tuple. If removing the tuple causes the count of b_j to drop to zero, `Revise` adds all of its piecewise functional blocks to the queue (i.e., blocks in other relations which were supports of b_j). This process repeats until quiescence. Note that Algorithm 2 is functionally identical to the one presented by Samaras and Stergiou [2005], albeit with different semantics to incorporate the changes related to partitioning relations according to subscopes. The asymptotic time and space complexity are also unchanged from their original publication describing PW-AC, as in the worst case the number of subscopes in the dual graph is equivalent to the number of constraints in the problem.

To use PW-AC2 during search, tuples are filtered from relations after an assign-

Algorithm 2: PW-AC2

```
1  Method PW-AC2($g$: A DualGraph of the CSP):
2      $g \leftarrow$ makeMinimal($g$)
3      initStructs($g$)
4      $Q_{cp} \leftarrow \emptyset$
5      foreach subscope $\sigma_i \in g$.subscopes do
6          foreach $v_i \in$ tupleBlocks[$\sigma_i$] do
7              if $\exists R_i$ s.t. incident($\sigma_i, R_i$) $\wedge$ tupleBlocks[$\sigma_i$][$v_i$][$R_i$] $= nil$ then
8                  foreach $R_i$ s.t. incident($\sigma_i, R_i$) $\wedge$ tupleBlocks[$\sigma_i$][$v_i$][$R_i$] $\neq nil$
                    do
9                      $Q_{cp} \leftarrow Q_{cp} \cup \{$tupleBlocks[$\sigma_i$][$v_i$][$R_i$]$\}$
10 Method Propagate($Q_{cp}$: A queue of tuple blocks):
11     while $Q_{cp} \neq \emptyset$ do
12         tupleBlock $\leftarrow$ pop($Q_{cp}$)
13         $\Delta \leftarrow$ Revise(tupleBlock)
14         if LivingTuples($R_i$) $= 0$ then
15             return Inconsistent
16         $Q_{cp} \leftarrow Q_{cp} \cup \Delta$
17     return Consistent
18 Method Revise($b_i$): A Tuple Block):
19     $\delta \leftarrow \emptyset$
20     foreach $\tau_i \in b_i$ do
21         if alive ($\tau_i$) then
22             alive ($\tau_i$) $\leftarrow false$
23             foreach subscope $\sigma_i$ s.t. incident($\sigma_i$, $R_i$) do
24                 $b_j \leftarrow$ blockLookup[$R_i$][$\sigma_i$][$\tau$]
25                 $b_j$.count $\leftarrow b_j$.count $-1$
26                 if $b_j$.count $= 0$ then
27                     foreach $block \in b_j$.pwctable do
28                         $\delta \leftarrow \delta \cup \{block\}$
29     return $\delta$
```

ment of a value to a variable (removing those tuples from relations that are incompatible with the assignment). As these tuples are removed, each block containing those tuples have their counts decremented. If a block's count drops to zero, its piecewise functional blocks are added to the queue just as they were in `Revise`. After propagation is complete, the modified relations are projected onto the variables in their constraints' scopes to remove any values from the domains of variables who no longer have supporting tuples in the relation.

4.4 Empirical Evaluation of PW-AC2 and eSTR2$^{(w)m}$

In addition to PW-AC2, we also evaluate variants of the eSTR family of algorithms introduced by Lecoutre *et al.* [2013] that use the minimal dual graph, and are denoted as eSTR2$^{(w)m}$. The changes required to enforce fPWC using the minimal dual graph in eSTR are straightforward – literally only a single line that computes the minimal dual graph, which is then used in the construction of the various support structures used in the algorithm. The correctness of the algorithm is guaranteed due to the properties of enforcing PWC on minimal dual graphs as outlined by Janssen *et al.* [1989].

The experiments were run on 48 benchmarks of non-binary CSPs. These benchmarks were selected from the CPAI08 dataset[1], and were limited to non-binary benchmarks with at least one instance with a subscope of cardinality greater than one (that is, benchmarks with non-trivial subscopes). Benchmarks with only trivial subscopes were excluded as enforcement of fPWC on these problems can be achieved using only GAC. This resulted in a total of 1,351 instances. In this section, we focus on seven algorithms: PW-AC, eSTR2, eSTR2^w, and PW-AC2^f, which enforce fPWC using the full dual graph, and their counterparts eSTR2^m, eSTR2^{wm}, and PW-AC2 which use the minimal dual graph to enforce fPWC.

Search over each CSP was performed using STAMPEDE to find the first consistent

[1]http://www.cril.univ-artois.fr/CPAI08/

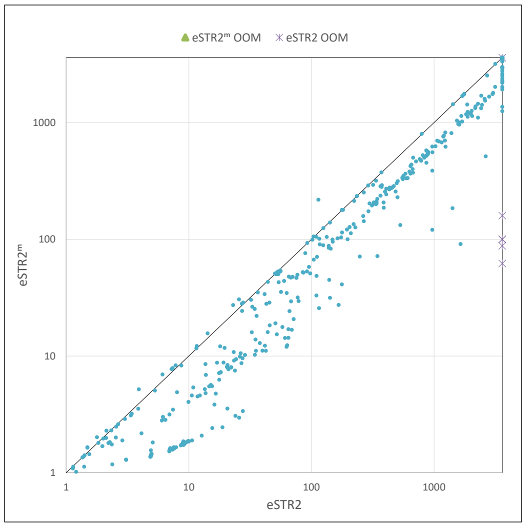
solution, using the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. The solver was limited to one hour and 8GB of memory. When an algorithm timed out or ran out of memory (OOM), it is recorded as having taken the full hour. We chose to focus on CPU time as our metric of comparison in *all* chapters of this dissertation. Metrics such as the number of constraint checks or node visits can be deceiving when comparing algorithms which enforce different levels of consistency. For example, with strong inference algorithms can dramatically reduce the number of nodes visited during search, perhaps even enforcing minimality, but would require much larger CPU overhead due to the additional reasoning required.

We begin by demonstrating the strength of using the minimal dual graph when enforcing PWC. Figure 4.4 shows scatter charts for algorithms eSTR2, eSTR2^w, and PW-AC2. Each point in the chart corresponds to a CSP instance solved with each algorithm, where the x and y coordinates of the points are the time required to solve the instance for the full and minimal dual graph variants of each algorithm, respectively. When a point falls below the diagonal on the chart, it means the minimal dual variant of the algorithm was able to solve that particular instance faster. The axes are logarithmic to clarify the difference in performance on both easy and hard problems.

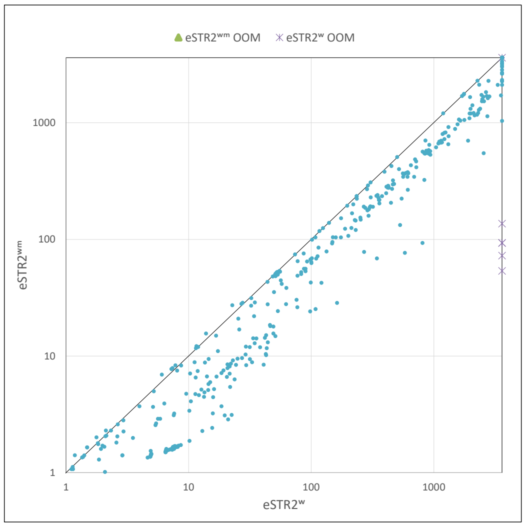
Table 4.1: Number of instances that ran out of memory per PWC algorithm.

Algorithm:	PW-AC	PW-AC2^f	PW-AC2	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2^m
# OOM:	148	132	132	138	138	132	132

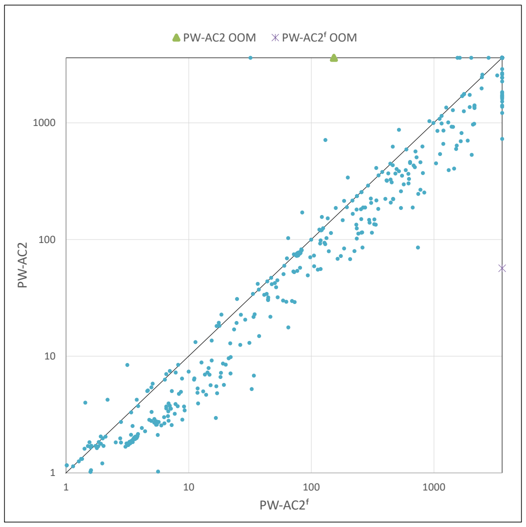
In each case, the version of the algorithm using the minimal dual graph is superior. Not only are the minimal dual versions able to solve instances consistently faster than their full dual graph counterpart, but they generally incur fewer losses due to memory consumption surpassing the 8GB limit as well. The number of instances that ran out of memory for each algorithm are listed in Table 4.1.



(a) eSTR2



(b) eSTR2^w



(c) PW-AC2

Figure 4.4: Pairwise comparisons between versions of PWC algorithms and their minimal dual counterparts using the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. Points below the line correspond to instances where the minimal dual variant was faster. Note the logarithmic scale.

PW-AC2 and PW-AC2^f run out of memory on the same number of problems; while operating and creating data structures based on the minimal dual graph has clear advantages, there's a small upfront cost when computing and removing the redundant edges. The increase in memory required while computing the minimal dual graph is transitory, but in some cases is enough to push the overall memory consumed over the limit. On the other hand, using the full dual graph requires persistent additional memory consumption due to the redundant subscopes and tuple blocks created, *as well as* an increase in time required to solve the instance.

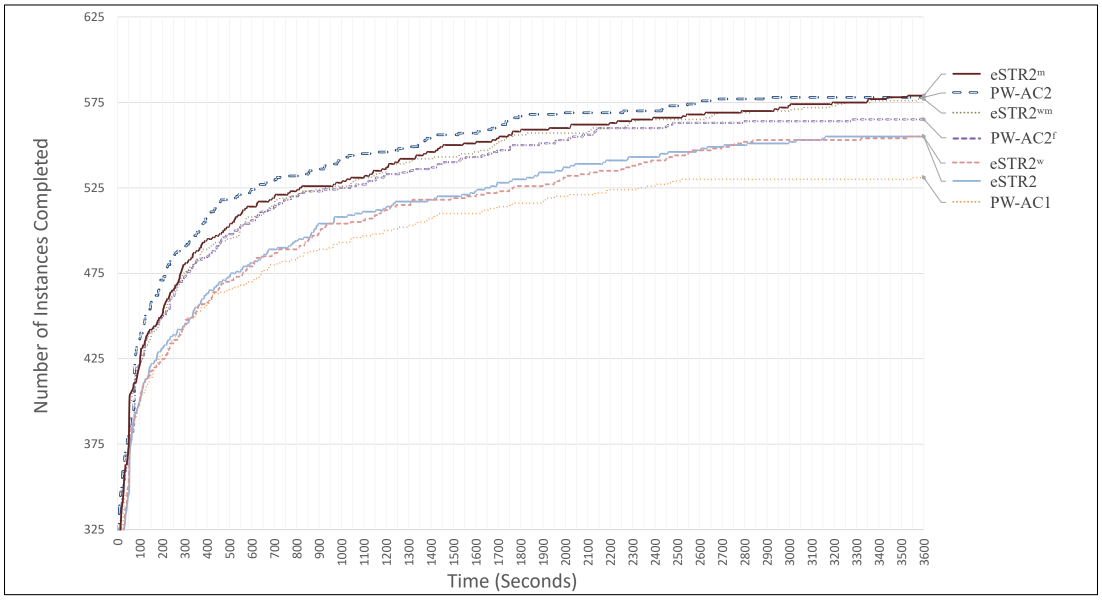


Figure 4.5: Cumulative chart showing number of instances completed in N seconds for PWC algorithms and their minimal dual counterparts using the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ heuristic.

Figure 4.5 provides a cumulative chart showing the number of instances able to be solved on each tested algorithm within a certain amount of time, up to the limit of one hour. All three algorithms that were based on the minimal dual graph outperform the others. We note again that eSTR2 algorithms had previously been formulated only using the full dual graph. PW-AC2 performs very well, though eSTR2^m is competitive and eventually succeeds at solving one more instance than PW-AC2.

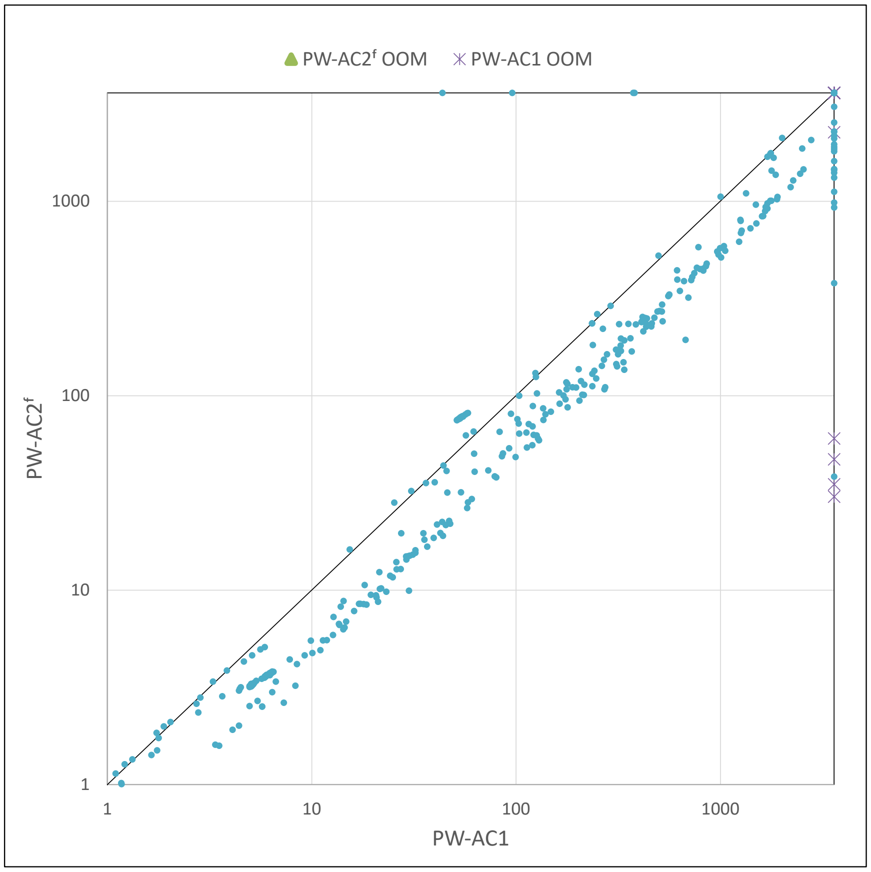


Figure 4.6: Pairwise comparison between PW-AC2 using the full dual graph and PW-AC.

PW-AC lags behind, though this is somewhat expected; of the properties outlined in Section 4.2, PW-AC only incorporates piecewise functionality. The eSTR2^{m} and eSTR2^{wm} algorithms have all of the desirable properties mentioned in Section 4.2 with the exception of replacing pairwise reasoning with a subscope-centric driven propagation mechanism. Figure 4.6 shows the strength of a subscope-centric approach. It compares the relative performance of PW-AC and PW-AC2^{f}; the difference between these two algorithms lays mainly in the subscope-based partitioning and their respective data structures, as the algorithms used to propagate are functionally identical. Clearly there is a large advantage to subscope-based partitioning in terms of both CPU time and memory consumption.

Table 4.2: Pairwise t-test results for tested PWC algorithms

	PW-AC	PW-AC2^f	PW-AC2	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2^m
PW-AC	-	-	-	-	-	-	-
PW-AC2^f	T	-	-	T	T	-	-
PW-AC2	T	T	-	T	T	T	T
eSTR2	T	-	-	-	T	-	-
eSTR2^w	T	-	-	-	-	-	-
eSTR2^{wm}	T	-	-	T	T	-	-
eSTR2^m	T	T	-	T	T	T	-

Table 4.2 shows a pairwise t-test on the runtime of all instances between all tested algorithms. The cells are labeled "True" if the population means show a statistically significant difference for the two tested algorithms ($p < .05$) and the average runtime for the algorithm in that row was less than the average runtime for the algorithm in that column. Here we can see that the minimal versions of all algorithms outperformed any other, and PW-AC2 is the best performing algorithm.

These results show the strengths of reasoning based on subscopes (PW-AC2, PW-AC2^f), the minimal dual graph (PW-AC2, eSTR2^m, eSTR2^{wm}), and selective enforcement of PWC (eSTR2, eSTR2^w, eSTR2^m, eSTR2^{wm}).

However, relative to algorithms for enforcing GAC that were contemporary with the tested algorithms, PWC still lags far behind. Figure 4.7 shows the same cumulative chart as Figure 4.5 but with the inclusion of STR2 and GAC2001, two algorithms used to enforce GAC.

Figure 4.8 shows a pairwise comparison between PW-AC2 and GAC2001. While there are select instances where PW-AC2 outperforms STR2, the additional memory consumption incurred by block creation in PW-AC2 causes a large number of OOMs relative to STR2. Additionally, while GAC is a relatively weak consistency, it is

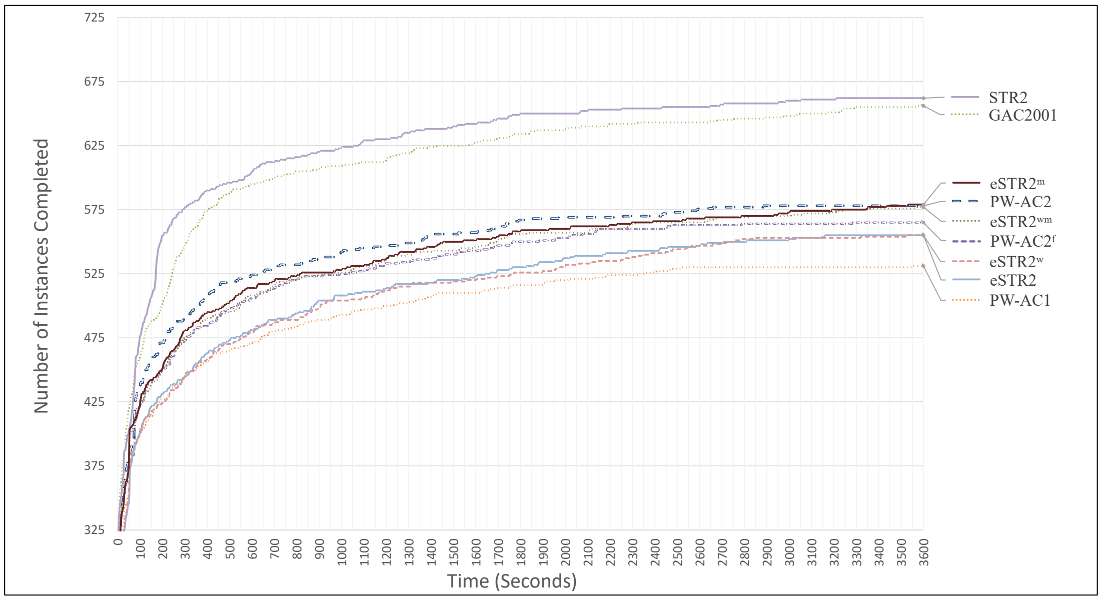


Figure 4.7: Cumulative chart showing number of instances completed in N seconds for GAC and PWC algorithms that were contemporaneous with PW-AC2.

cheap to enforce, and if it is capable of finding an inconsistency at a particular search node, any additional effort that would have been made to enforce PWC at the same node would be wasted.

4.5 PW-CT: Efficiently and Lazily Enforcing Pairwise Consistency

We now introduce PW-CT, an algorithm for enforcing fPWC, as an extension of CompactTable [Demeulenaere *et al.*, 2016]. PW-CT requires few modifications to the original CT structures, exploits mechanisms from existing PWC algorithms, and integrates additional improvements discussed previously in this chapter. More specifically, PW-CT uses CompactTable (i.e., GAC) as much as possible to avoid costly PWC checks in two ways: by ensuring the problem is GAC before resorting to any PWC checks and by identifying situations where GAC guarantees PWC. Finally,

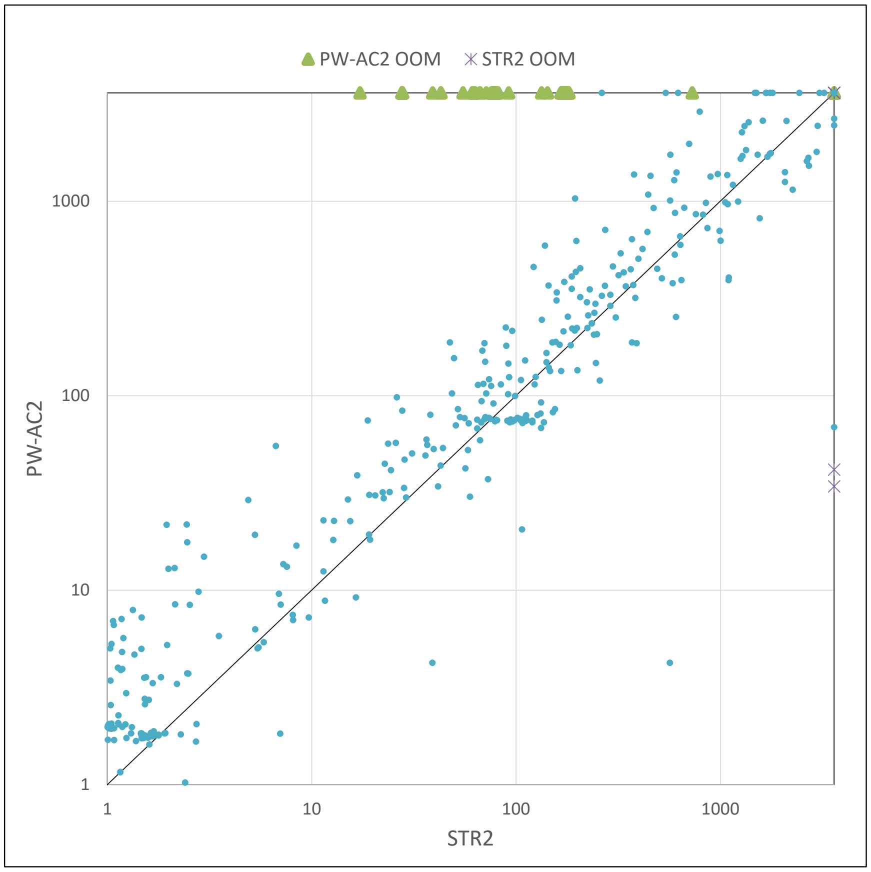


Figure 4.8: Pairwise comparison between PW-AC2 and STR2.

it exploits properties of the dual encoding of the CSP to speed-up processing and reduce memory consumption.

First, we provide a brief overview of CompactTable, before proceeding to the description and experiments of PW-CT.

4.5.1 A Brief Overview of Algorithm CompactTable

CompactTable [Demeulenaere *et al.*, 2016] is an algorithm that enforces GAC on a problem in a manner similar to the tabular reduction methods used in the STR family of algorithms. The main idea of these algorithms is to track which variables have been modified since the last time GAC was enforced, filter tuples in the table constraints whose scopes contain the modified variables, and update the domains of the variables in the constraints' scopes based on the recently removed tuples. Various support structures are used to make each phase of these algorithms more efficient.

In the case of CompactTable, these support structures take the form of reversible, sparse bitsets. There are two types of bitsets created for CompactTable: the first is a bitset that corresponds to the currently living tuples in each constraint. Each bit in the set corresponds to the ID of a tuple in the constraint. The bit is set if the tuple is alive and unset if it has been removed from the constraint. The second type of bitset is static throughout the course of search. These sets are created for each combination of variable-value pair in the scope of a constraint, and correspond to tuples that contain the given variable-pair. Thus, both types of sets are the same size, and will contain one bit per tuple in a given constraint.

Updating a table constraint to remove invalid tuples due to variable-value deletions can be accomplished by intersecting the bitset corresponding to the living tuple in the constraint with the bitset of the tuples that contained that variable-value pair. Determining if the removal of tuples caused another value to lose its support can be done by checking if the intersection of supports for a given variable-value pair and the living tuples in the constraint is the empty set.

4.5.2 Data Structures

Support Structures: Both CompactTable and PW-CT represent the living tuples in a constraint as an RSparseBitSet. The RSparseBitSet stores four members: an array of reversible 64-bit integers called words,[2] a reversible integer called limit that represents the number of non-zero integers in words, an array called index that stores the position of all non-zero integers in words in locations less-than or equal-to limit,

[2]64-bit on most current architectures.

and an array called mask used to modify the set. Demeulenaere *et al.* [2016] introduce member functions of the RSparseBitSet used by PW-CT which we briefly review: function addToMask takes an array and alters mask to be the bitwise OR of the array and the current mask, function intersectWithMask alters words to be the bitwise AND of the current words and mask, and function clearMask sets the integers in mask to 0.

The RSparseBitSet for a constraint c_i is denoted as living(c_i). The data structure supports[c_i, x, a] is a static array of bits corresponding to the tuples of a constraint c_i that have the value a for variable x.[3] To improve performance of various functions in PW-CT, we introduce a structure indices[c_i, x, a], which is an RSparseBitSet that stores the positions in supports[c_i, x, a] that are non-zero.

PW-CT uses two maps. The first, incidentCons[σ], gives the list of constraints incident to a non-trivial subscope σ. The second, incidentSubscopes[c_i], gives the list of non-trivial subscopes incident to a constraint c_i. We can optionally use the minimal dual graph to reduce the number of generated subscopes in each map without affecting the level of consistency enforced (see Section 4.2). Importantly, all these support structures are created at initialization.

Blocks: We represent a block as a simple structure with a member sets, which is a vector of pointers to supports[c_i, x, a] representing the signature of the block, and a member commonIndices, which is an RSparseBitSet of the indices shared by all of the supports in sets. Performing an intersection of the sets in a block computes the set of tuples with the signature corresponding to sets. In PW-CT, blocks are never stored but always computed dynamically during search.

[3]Note that we have added the additional parameter c_i to supports[] to uniquely determine the constraint's supports we are referring to in the pseudocode.

Algorithm 3: CreateBlock(c_i, τ, σ)

```
Input: A constraint $c_i$, a tuple $\tau$, and a subscope $\sigma$
Output: A block $b$
1 $j \leftarrow 0$
2 foreach variable $x \in \sigma$ do
3     $b$.sets[$j$] $\leftarrow$ supports[$c_i$][$x$][$\tau[x]$]    // $\tau[x]$ is the value for $x$ in tuple $\tau$
4     ind[$j$] $\leftarrow$ indices[$c_i$][$x$][$\tau[x]$]
5     if ind[$j$].limit $<$ ind[0].limit then swap(ind[$j$], ind[0])
6     $j \leftarrow j + 1$
7 $b$.commonIndices.initIntersection(ind)
8 return $b$
```

The function CreateBlock (Algorithm 3) takes as input a constraint, tuple, and subscope and returns a block structure, which can be used to dynamically compute the partition of tuples of the constraint with the corresponding signature. The RSparseBitSet commonIndices improves performance of some operations of the methods listed in Algorithm 4. Note that the method `initIntersection` called in Line 7 is defined in Algorithm 4 and makes use of the call `swap` in Line 5.

Additional Methods for the RSparseBitSet Class: Algorithm 4 introduces additional methods for the RSparseBitSet class for use in PW-CT. The method `initIntersection` is used in CreateBlock to initialize the RSparseBitSet with the indices common to a collection of RSparseBitSets.

Algorithm 4: Additional algorithms required for RSparseBitSet

```
Method initIntersection(sets: A vector of RSparseBitSets):
    limit ← $-1$
    index ← $\emptyset$
    Expand words and index to size of sets[0].words
    foreach $i \leftarrow 0$ to sets[0].limit do
        offset ← sets[0].index[$i$]
        bits ← sets[0].words[offset]
        for set ∈ sets and bits $\neq 0$ do
            bits ← bits & set.words[offset]                    // Bitwise AND
        if bits $\neq 0$ then
            words[offset] ← bits
            limit ← limit + 1
            index[limit] ← offset
Method intersectIndex(block: A Block created by CREATEBLOCK):
    // If limit < block.commonIndices.numSet(), iterate from 0 to
        limit
    for offset ∈ block.commonIndices do
        intersection ← words[offset]
        for set ∈ block.sets and intersection $\neq 0$ do
            intersection ← intersection & set.words[offset]     // Bitwise AND
        if intersection $\neq 0$ then return offset
    return -1
Method removeBlock(block: A Block created by CREATEBLOCK):
    for $i \leftarrow$ limit to 0 do
        offset ← index[$i$]
        if offset ∈ block.commonIndices then
            b ← 64-bit Integer with all bits set
            for set ∈ block.sets and b $\neq 0$ do
                b ← b & set.words[offset]                      // Bitwise AND
            words[offset] ← words[offset] & ~b                 // Bitwise NOT
            if words[offset] = 0 then
                index[$i$] ← index[limit]
                index[limit] ← offset
                limit ← limit $-1$
```

We overload the original RSparseBitSet method intersectIndex to operate on blocks. It is similar in behavior to the original intersectIndex, differing in that it determines if a *block* of tuples has a support in the set, rather than a single variable-value pair. The method removeBlock computes the set-difference between the RSparseBitSet and a block of tuples. Anytime the bits at position offset in the words member is modified, the RSparseBitSet computes the difference in set bits for the integer at words[offset]. Method numSet returns the total number of set bits in the RSparseBitSet. This operation incurs some overhead but is needed in PW-CT.

There were a few functions omitted from the pseudocode in Algorithm 4 for brevity. The methods save and restore respectively save and restore the state of the reversible elements in the RSparseBitSet and follow from the definition of a reversible set. We maintain the number of living bits when altering the set and numSet returns this value.[4] PW-CT relies on the ability to discover the tuples removed between two points in time (the delta of the set). To this end, method computeDelta returns an RSparseBitSet containing the bits removed between the current state of the RSparseBitSet and the last stored state. Method clearDelta readies the set to track the next set of removed tuples, but does not alter the currently set bits. These were implemented using the method save and comparing the reversible primitives of the current state of the set and its previously saved state. Method addBlockToMask behaves like the original addToMask, but adds to the mask only those bits common to all bitsets in the block. Its implementation follows from addToMask and intersectIndex. We also assume that the bits in the RSparseBitSets are iterable and treat the bits and the tuples they represent interchangeably in our pseudocode for simplicity.

[4]This can be done efficiently in C++ with Clang/GCC's builtin popcountll function.

4.5.3 Enforcing PW-CT

Roughly speaking, PW-CT has two main phases: a GAC phase, in which CompactTable is executed until quiescence, and a PWC phase that performs a *single* pass over the tuples deleted by CompactTable to uncover new non-PWC blocks. PW-CT maintains two queues: CTQueue tracks constraints that must be 'checked for GAC' and PWCQueue tracks constraints that have lost tuples, thus threatening the PW-consistency of blocks in other constraints. For brevity we assume both queues act as sets in that there is at most one instance of a particular constraint in the queue at a time.

Algorithm 5: Lookahead($\mathcal{P}$) Enforces PWC on a CSP $\mathcal{P}$

```
Input: A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$
Output: Whether the current problem is consistent
1  consistent ← true
2  if $\mathcal{P}$ has not been preprocessed then
3      consistent ← PreProcess($\mathcal{C}$)
4  while consistent and not empty(CTQueue) do
5      $c_i$ ← pop(CTQueue)
6      consistent ← CompactTable($c_i$)
7      if $c_i$ was modified then push($c_i$,PWCQueue)
8      if consistent and empty(CTQueue) then
9          mCons ← PWCQueue
10         for $c_i \in$ mCons and consistent do
11             consistent ← EnforcePWC($c_i$)
12             living($c_i$).clearDelta()
13             PWCQueue ← PWCQueue $\setminus \{c_i\}$
14 return consistent
```

Function Lookahead (Algorithm 5) is the entry point for PW-CT. Lines 4 to 7 run CompactTable until quiescence and enqueues constraints modified by GAC into PWCQueue. CompactTable enqueues constraints with modified variables into CTQueue, thus, when execution hits Line 9, the problem is GAC but not necessarily

PWC. Lines 9 to 13 call function EnforcePWC (Algorithm 6) on modified constraints to determine if the removal of tuples in each constraint c_i in the queue causes the loss of a PW-support in another constraint.

Algorithm 6: EnforcePWC(c_i) Propagates invalid blocks of c_i

```
Input: Constraint $c_i$ that has been modified by CT
Output: Whether the current problem is consistent
1  tupsToCheck ← living($c_i$).computeDelta()
2  for $\sigma \in$ incidentSubscopes[$c_i$] do
3      tupsToCheck.save()
4      for variable $x \in \sigma$ s.t. $x$ was modified on previous call to CT do
5          if $|\mathcal{D}(x)| < |\Delta_x|$ then
6              tupsToCheck.clearMask()
7              for value $a \in \mathcal{D}(x)$ do
8                  tupsToCheck.addToMask(supports[$c_i$][$x$][$a$])
9              tupsToCheck.intersectWithMask()
10         else
11             for value $a \in \Delta_x$ do
12                 $b$ ← an empty block
13                 $b$.sets ← supports[$c_i$][$x$][$a$]
14                 $b$.indices ← indices[$c_i$][$x$][$a$]
15                 tupsToCheck.removeBlock($b$)
16     for $\tau \in$ tupsToCheck do
17         consistent ← ReviseBlock($c_i, \sigma, \tau$)
18         if not consistent then return false
19         tupsToCheck.removeBlock(CreateBlock($c_i, \sigma, \tau$))
20     tupsToCheck.restore()
21 return true
```

EnforcePWC iterates over all subscopes incident to a constraint and the constraint's most recently removed tuples, checking whether the block induced by the combination of each subscope and tuple is empty. As discussed in Section 4.2.4, any blocks whose signatures have variable-value pairs removed by GAC necessarily have had all of their supporting blocks in neighboring constraints removed as well. The loop beginning at Line 4 in EnforcePWC (Algorithm 6) takes advantage of this insight by discarding blocks of tuples from consideration for PW-support checks, skipping unnecessary calls to ReviseBlock (Algorithm 7). It uses a mechanism similar to incremental and reset-based updates [Perez and Régin, 2014], where Δ_x is the set of values of variable x removed by the previous call to CompactTable.

```
Algorithm 7: ReviseBlock($c_i, \sigma, \tau$)                Removes supports of empty block
   Input: A constraint $c_i$, a subscope $\sigma$, and a tuple $\tau$
   Output: Whether the current problem is consistent
1  if living($c_i$).intersectIndex(CreateBlock($c_i, \sigma, \tau$)) = -1 then
2      for $c_j \in$ incidentCons[$\sigma$] s.t. $c_i \neq c_j$ do
3          living($c_j$).removeBlock(CreateBlock($c_j, \sigma, \tau$))
4          if living($c_j$) was modified then
5              if living($c_j$).numSet() = 0 then return false
6              push($c_j$,PWCQueue)
7              push($c_j$,CTQueue)
8  return true
```

Lines 16 to 19 check the block induced by each removed tuple for the current subscope for validity by calling function ReviseBlock (Algorithm 7). If no other tuples in the induced block are alive in the constraint, ReviseBlock removes the piecewise-functional blocks from all other constraints incident to the current subscope, and enqueues the constraints modified during this process. Multiple tuples in the set of removed tuples may belong to the same block for a given subscope, so, Line 19 removes all other tuples from that block from the set of tuples to check (as successive calls for the same block would be redundant).

It is advantageous to interleave CompactTable and EnforcePWC calls because tuples removed by EnforcePWC may enable value deletions that can be propagated quickly by CT. To prevent running EnforcePWC until quiescence on the first pass, a copy of the queue is created in Line 9 of Lookahead (Algorithm 5). As a result, each modified constraint is processed at most once at each PWC pass.

Proposition 3. *If the CSP is initially PWC,* LOOKAHEAD *guarantees fPWC.*

Proof. Consider a constraint c_i altered by COMPACTTABLE. Because the problem was PWC prior to running COMPACTTABLE, the only 'endangered' blocks in c_i have tuples deleted by COMPACTTABLE. To enforce PWC, we need to check if any block b_i whose signature is a combination of a deleted tuple τ of c_i, a subscope σ_i incident to c_i and c_i is empty as a result of COMPACTTABLE. If we find a block b_i to be empty, we can remove the blocks that are PW-supports of b_i from all constraints c_j incident to σ_i. Because each c_j modified in REVISEBLOCK is added to the `PWCQueue` (Line 6), the removal of any tuple in c_j by REVISEBLOCK that emptied a block induced on any subscope σ_j is necessarily detected by the next call to ENFORCEPWC(c_j). Running COMPACTTABLE in between calls to ENFORCEPWC on any modified constraint ensures that the domains of the variables in the scope of the constraint are 'synced' with the constraint's relation, thus, ensuring fPWC. □

Proposition 4. *The time complexity of calling* ENFORCEPWC *on a constraint is* $O((|\mathcal{C}| \cdot t) \cdot (\lceil \frac{t}{64} \rceil \cdot |\mathcal{C}| + |\sigma|))$*, where t is the number of tuples in the largest constraint and σ the largest subscope.*

Proof. REVISEBLOCK iterates over the constraints incident to a subscope, which in the worst case is $|\mathcal{C}| - 1$. Each constraint may need to call `removeBlock`, which requires iterating over $\lceil t/64 \rceil$ elements. Creating the block requires iterating over σ. The only tuples evaluated by REVISEBLOCK are those that have been removed from a constraint, and at most t tuples can be removed. A removed tuple can be revised for each of its constraint's incident subscopes. In the worst case, a constraint has $|\mathcal{C}| - 1$ neighbors in the dual graph, and each neighbor induces a unique subscope. Therefore, each tuple in the problem may cause REVISEBLOCK to be called $O((|\mathcal{C}| \cdot t) \cdot (\lceil \frac{t}{64} \rceil \cdot |\mathcal{C}| + |\sigma|))$ times. □

Algorithm 8: PreProcess(C) Runs CT and removes non-PWC tuples

```
Input: A set of constraints $C$
Output: Whether the current problem is consistent
1 Run CompactTable() until quiescence
2 if consistent then
3     consistent ← InitPWC($C$)
4     if consistent then
5         forall $c_i \in C$ do living($c_i$).clearDelta()
6         consistent ← InitPWC($C$)
7 return consistent
```

PW-CT requires an additional initialization step to guarantee that preprocessing enforces fPWC. Consider the tuple $\tau = \{\langle A,1\rangle,\langle B,1\rangle,\langle C,1\rangle, \langle E,1\rangle\}$ in R_1 in Figure 4.1. Each variable-value pair in τ has a GAC support in R_2, but no PW-support. EnforcePWC operates on deleted tuples in order to propagate PW-support removals, but because τ's variable-value pairs are GAC, τ is not deleted by CompactTable. Therefore, EnforcePWC is not called. To remedy this situation, all blocks that initially lack PW-supports need first to be removed from the problem. Once this removal is done, EnforcePWC can then evaluate the deleted tuples and propagate any other blocks that are emptied by their removal. Function PreProcess (Algorithm 8) accomplishes this operation by first enforcing GAC with CompactTable and then calling function InitPWC (Algorithm 9).

Function InitPWC considers each subscope σ in the problem. It begins by finding the constraint c_s with the smallest number of living tuples incident to σ. The algorithm checks if the blocks induced by the subscope σ for each tuple τ in `living`(c_s) has a PW-support in all constraints incident to σ (Lines 6 to 9). If the block is supported, then, for each constraint c_j incident to σ, we add the block of tuples induced by τ to the mask of `living`(c_j) (Line 12). After all blocks of c_s are processed, the masks of each `RSparseBitSet` `living`(c_j) contain only PW-supported tuples. Line 15 removes

Algorithm 9: InitPWC(C) Partially enforces PWC on the constraints

```
   Input: A set of constraints $C$
   Output: Whether the problem is consistent at preprocessing
1  for $\sigma \in$ Subscopes do
2      $c_s \leftarrow$ constraint with fewest living tuples $\in$ incidentCons[$\sigma$]
3      foreach $c_i \in$ incidentCons[$\sigma$] do living($c_i$).clearMask()
4      toCheck $\leftarrow$ living($c_s$)                       // Makes a copy
5      for $\tau \in$ toCheck do
6          tuplePWC $\leftarrow$ true
7          foreach $c_i \in$ incidentCons[$\sigma$] and tuplePWC do
8              if living($c_i$).intersectIndex(CreateBlock($c_i, \tau, \sigma$))$= -1$ then
9                  tuplePWC $\leftarrow$ false
10         if tuplePWC then
11             foreach $c_i \in$ incidentCons[$\sigma$] do
12                 living($c_i$).addBlockToMask(CreateBlock($c_i, \tau, \sigma$));
13         toCheck.removeBlock(CreateBlock($c_s, \tau, \sigma$))
14     foreach $c_i \in$ incidentCons[$\sigma$] do
15         living($c_i$).intersectWithMask()
16         if living($c_i$) was modified then
17             push($c_i$,PWCQueue)
18             push($c_i$,CTQueue)
19         if living($c_i$).numSet $= 0$ then return false
20
21 return true
```

non-PWC tuples by calling the `intersectWithMask` method for each `living`(c_j). In practice, we found that in some problems the number of initially non-PWC tuples can be extremely large causing significant slowdown in the first call to EnforcePWC after PreProcess. To alleviate some of this burden, Function PreProcess runs InitPWC twice. The second call to InitPWC tends to remove fewer tuples than the first and guarantees the removal of any block that lost PW-supports due to the first call. Note that only the first call is strictly necessary for correctness.

4.6 Empirical Evaluation of PW-CT

As with Section 4.4, the experiments were run on 48 benchmarks of non-binary CSPs and were limited to non-binary benchmarks with at least one instance with a non-trivial subscope. This resulted in a total of 1,351 instances. In this section, we focus on eight algorithms: PW-AC2, PW-AC2-CT, PW-CT, PW-CT^f, CompactTable, STRbit, GAC2001, and STR2. In Section 4.4 we showed the strength of PW-AC2 relative to other algorithms used to enforce fPWC, so we use PW-AC2 as a baseline of fPWC performance. We also include a version of PW-AC2, PW-AC2-CT, which runs the CompactTable algorithm prior to enforcing PW-AC2 to demonstrate that naively enforcing a GAC prior to enforcing PWC does not provide the same performance as interleaving them in an algorithm like PW-CT. CompactTable and STRbit are both algorithms that enforce GAC using bitsets, but their data structures and methods differ. Algorithms STR2 and GAC2001 represent the "classical" method of enforcing GAC both with and without tabular reduction, respectively.

As before, search over each CSP was performed using Stampede to find the first consistent solution, using the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. The solver was limited to one hour and 8GB of memory. When an algorithm timed out or ran out of memory (OOM), it is recorded as having taken the full hour.

Figure 4.9 shows a cumulative chart providing the number of instance solved by the tested algorithms within a certain amount of time. We'll begin by delving into the relative performance of PW-CT and PW-CT^f. Surprisingly we can see that the version of PW-CT operating on the full dual graph outperforms the version operating on the minimal dual graph. The cause of this becomes clear when we examine the scatter plot in Figure 4.10 for these two algorithms.

The large grouping of OOM instances on the right side of Figure 4.10 are all from

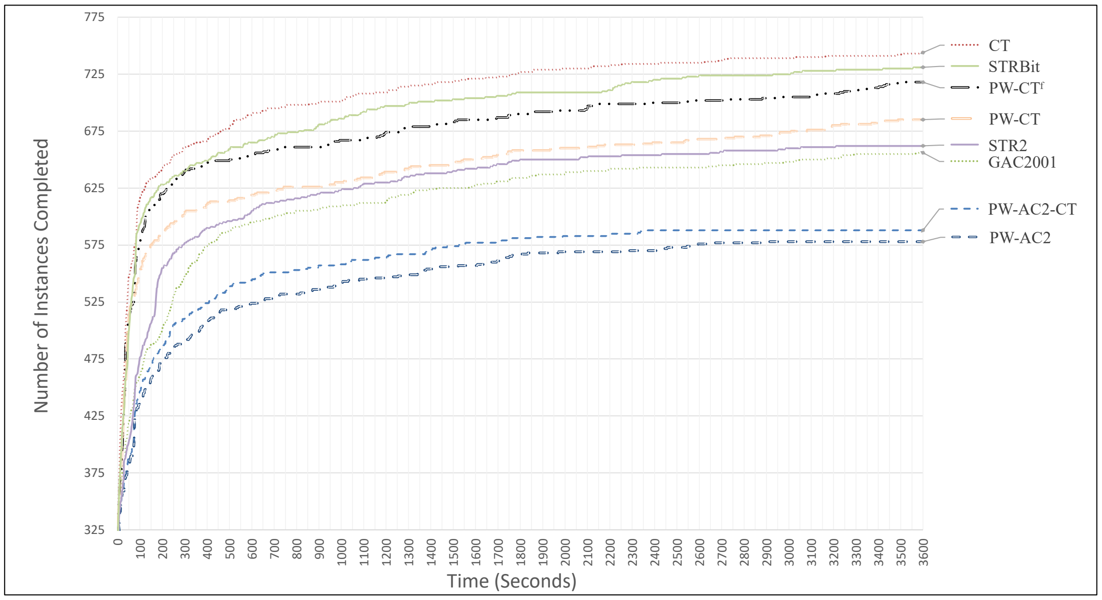


Figure 4.9: Cumulative chart showing number of instances completed in N seconds for GAC and select PWC algorithms.

a single benchmark, BddLarge. The instances in this benchmark are all structurally similar, with roughly 2,700 constraints that are all fully connected. Computing the minimal dual graph requires roughly 16GB of memory with our current implementation, and while removing redundant edges in the dual graph reduces the average degree of a node considerably (from 2,712 to an average of 60 and a maximum of 108 in the specific instance examined), it comes at a considerable cost in memory and CPU time, and leaves the total number of non-trivial subscopes in the problem unchanged (roughly 1.1 million). This benchmark (and its smaller version, BddSmall) are somewhat pathological for PWC algorithms. The high number of tuples in the problem cause the classical PWC algorithms presented earlier in this chapter to run out of memory regardless of whether the algorithm was operating on the minimal or full dual graph, but because PW-CT does not store *any* coarse blocks, it is uniquely impacted.

Figure 4.11 shows a cumulative graph with the BddLarge instances excluded.

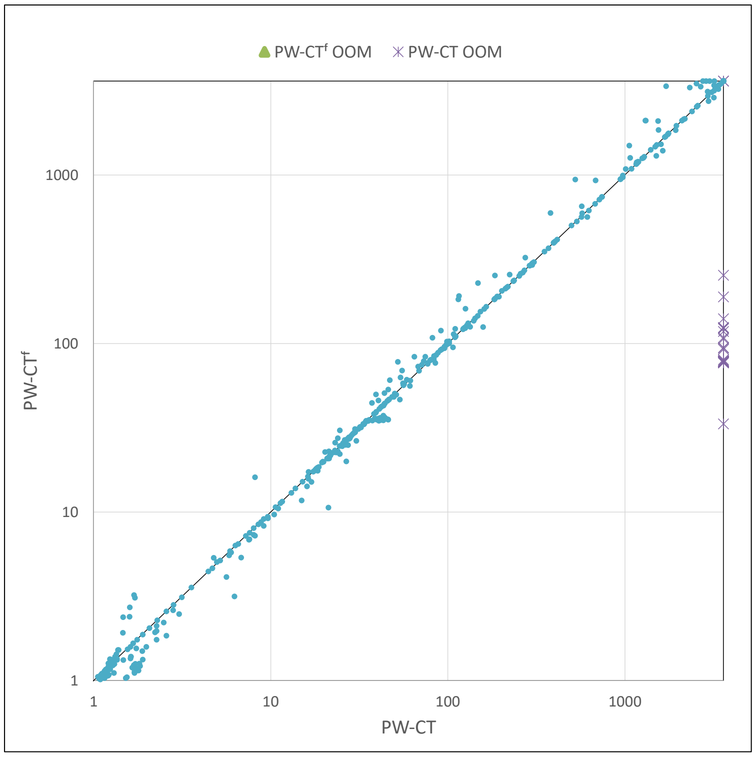


Figure 4.10: Scatter chart comparing minimal dual and full dual versions of PW-CT.

Here we can see that the full and minimal dual version of PW-CT are extremely competitive with the minimal dual version edging out the full dual version. The gains we see in other PWC algorithms from using the minimal dual graph are minimized with PW-CT for two reasons. The first is the relatively small number of times PW-CT enforces PWC at all, as it enforces GAC between each run of PWC. The second is the relatively small cost of enforcing PWC in PW-CT since the tuples in constraints are processed as 64-bit chunks.

Figure 4.12 shows scatterplots comparing PW-CT to two GAC algorithms: STR2 and CompactTable. STR2 is representative of the "classical" GAC algorithms while CompactTable is the current best-in-class for enforcing GAC. While PW-CT outperforms STR2 on the vast majority of instances, including some which STR2

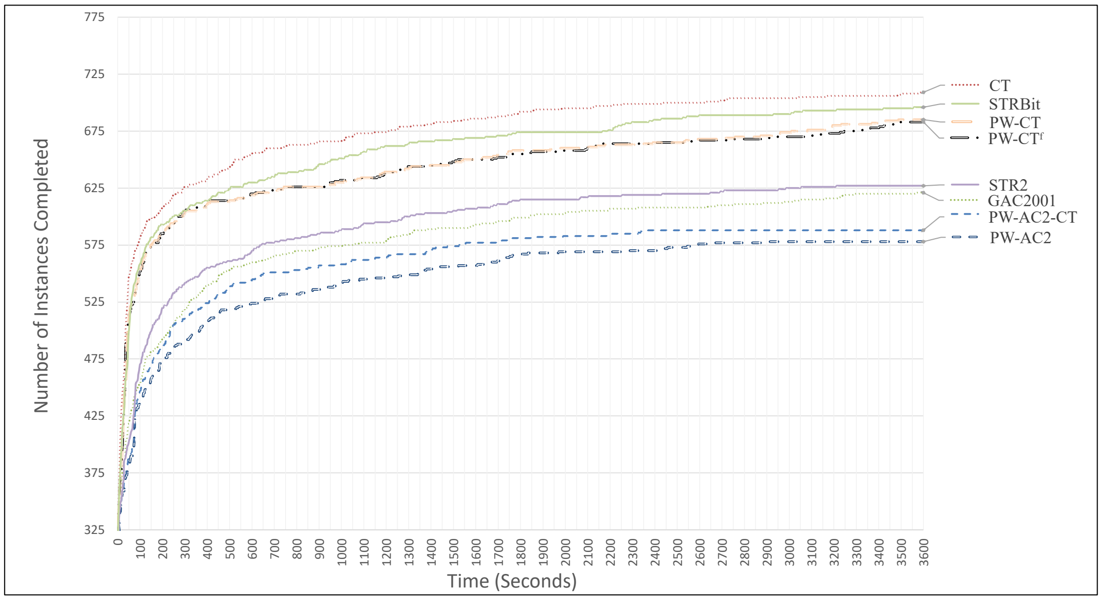
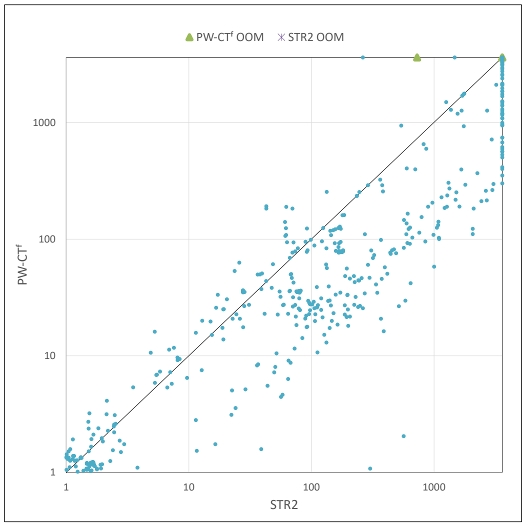


Figure 4.11: Cumulative chart showing number of instances completed in N seconds for GAC and select PWC algorithms excluding instances from the BDDLarge dataset.

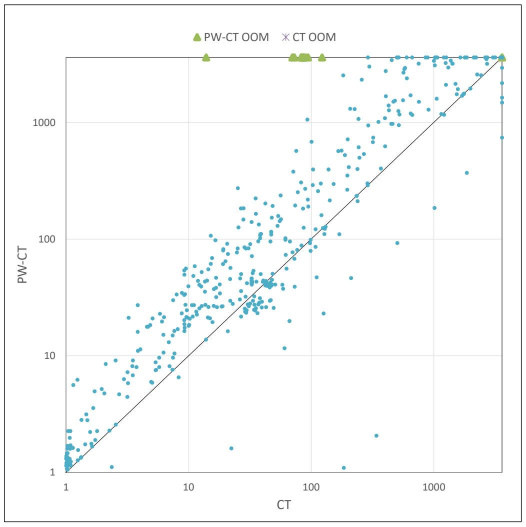
was unable to solve at all within the time constraints, CompactTable still handily outperforms PW-CT on all but a handful of instances.

Table 4.3: Pairwise t-test results for all tested PWC and GAC algorithms with $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. Cells are labeled "T" if the results were significantly different (p <.05) and the value of the row entry was less than the column.

	CT	GAC2001	PW-AC2-CT	PW-AC2	PW-CT^f	PW-CT	STR2	STRbit
CT	-	T	T	T	T	T	T	T
GAC2001	-	-	T	T	-	-	-	-
PW-AC2-CT	-	-	-	T	-	-	-	-
PW-AC2	-	-	-	-	-	-	-	-
PW-CT^f	-	T	T	T	-	T	T	-
PW-CT	-	T	T	T	-	-	-	-
STR2	-	T	T	T	-	-	-	-
STRbit	-	T	T	T	T	T	T	-



(a) PW-CT and STR2



(b) PW-CT and CompactTable

Figure 4.12: Pairwise comparisons between PW-CT and two GAC algorithms. Points below the line correspond to instances where PW-CT was faster.

This is further elucidated by the t-test results in Table 4.3.[5] However, PW-CT still represents a fundamental step forward for relational consistencies, and PWC in particular, as it is the first PWC algorithm we are aware of that is able to consistently outperform *any* classical GAC algorithm, and is still fairly competitive with modern GAC algorithms. We would also further emphasize that while this research has evaluated the use of HLC on full problems, their true strength lies in a targeted, localized approach [Woodward, 2018].

4.7 Future work

We would also be remiss to not mention the impact that variable ordering heuristics can have on search. While $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ has been shown to be generally more effective than $\frac{|\texttt{dom}|}{\texttt{ddeg}}$, the decisions made by $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ have previously been considered too unstable

[5]The results included in Table 4.3 include benchmark BddLarge; excluding BddLarge results in there being no statistical difference between PW-CT and PW-CT^f.

to objectively allow comparing algorithms' performance. Researchers studying the performance of HLC during search sometimes use $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ in their experiments [Balafrej *et al.*, 2015; Paparrizou and Stergiou, 2016, 2017]. We opted to use $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ in these experiments in the interest of showing performance akin to real-world use cases.

However, when looking at the number of nodes visited (i.e., search tree size) for algorithms CompactTable and PW-CT on the tested instances, 4.6% of the search trees were *larger* for PW-CT. Of those 4.6%, the average increase in the number of nodes visited was 95,328 (a 23.9% increase from CompactTable). While this occurred in only a small number of instances, that it happened at all given the increase in propagation strength PWC can provide is surprising. Ordering heuristics that are designed and tuned for relational consistencies, or robust to the propagation scheme being used, are an obvious next step in showing the efficacy of relational consistency algorithms.

Additionally, identifying strategies for selectively applying PWC when using PW-CT would likely prove to be beneficial. The algorithm is already very nearly on par with CompactTable and STRbit on many problems and exceeds the performance of other GAC and PWC algorithms on nearly every benchmark. Judicious application of PWC while maintaining GAC would more than likely produce a hybrid consistency that reliably exceeds the performance of CT. Such approaches have already been empirically validated, albeit with different choices for the HLC being used [Geschwender *et al.*, 2016; Karakashian *et al.*, 2011, 2013; Woodward *et al.*, 2017, 2018; Woodward, 2018].

Summary

In this chapter, we introduced two new algorithms for enforcing fPWC, PW-CT and PW-AC2, as well as modified versions of eSTR2 algorithms that use the minimal dual graph. We showed that PWC algorithms generally benefit from using a minimal dual graph to improve time and space cost. Further, we introduced three other ways to improve the efficacy of PWC algorithms, and designed an algorithm, PW-CT that uses all of them. Finally, we showed that the performance of PW-CT far and away exceeds that of other PWC algorithms, falling only behind the most recent GAC algorithms, STRbit and CompactTable.

Chapter 5

Improving m-wise Consistency Algorithms Via Dynamic Relation Partitioning

Chapter 4 introduced new algorithms and techniques for enforcing fPWC, which ensures that any tuple in one relation can be consistently extended to at least one tuple in another, effectively ensuring that any two relations are minimal with respect to one another. The obvious next step is ensuring minimality amongst an arbitrary number of relations. The consistency property R($*$,m)C [Karakashian *et al.*, 2010a, 2011, 2013] (originally known as m-wise consistency [Gyssens, 1986]) formally defines this property. In this chapter, we will provide a brief background of this consistency property and the algorithms that are used to enforce it, provide two new algorithms used to enforce m-wise consistency (previously published in [Schneider *et al.*, 2014]), and empirically evaluate the performance of these algorithms.

5.1 Background

We'll begin by reviewing the definition of PWC.

Definition 4. Pairwise Consistency (PWC) [Gyssens, 1986]: *A constraint network $\mathcal{P} = (\mathcal{X},\ \mathcal{D},\mathcal{C})$ is PWC iff, for every tuple t_i in every constraint c_i there is a tuple t_j in every constraint c_j such that $\pi_{subscope(c_i,c_j)}(t_i) = \pi_{subscope(c_i,c_j)}(t_j)$, t_j is called a PW-support of t_i in c_j. A CSP that is both PWC and GAC is said to be full PWC (fPWC).*

As mentioned in Chapter 4, the piecewise functionality property divides relations into equivalences classes of tuples (see Figure 5.1) and forms the basis of the PW-AC, PW-AC2, and PW-CT algorithms [Samaras and Stergiou, 2005; Schneider and Choueiry, 2018]. A natural extension of this property is extending the guarantee to m other constraints. We call this a "parameterized" consistency because the value of m is not fixed.

Definition 5. R($*$,m)C [Karakashian *et al.*, 2010a]: *A constraint network* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is R($*$,m)C iff every tuple in the relation of each constraint* $c_i \in \mathcal{C}$ *can be extended to the variables in* $\bigcup_{c_j \in \mathcal{C}} scope(c_j) \setminus scope(c_i)$ *in an assignment that satisfies all the constraints in* $\mathcal{C}$ *simultaneously. A network is R($*$,m)C iff every set of* m *constraints,* $m \geq 2$*, is R($*$,m)C.*

The consistency property R($*$,m)C ensures the minimality of every combination of m relations. It is equivalent to m-wise consistency defined in Databases [Gyssens, 1986]. The parameterized algorithm PerTuple enforces R($*$,m)C [Karakashian *et al.*, 2010a, 2013]. It ensures that each tuple in a relation appears in a solution of the dual CSP induced by any m relations by conducting a backtrack search on the tuples over the other $m-1$ relations in the combination (see Figure 5.2).

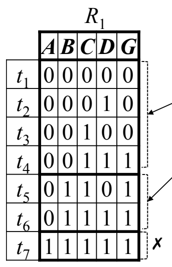


Figure 5.1: Piecewise functional constraint.

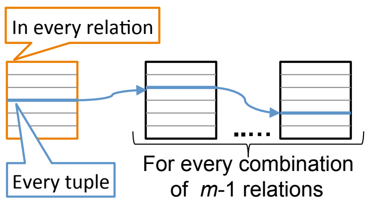


Figure 5.2: Illustrating R($*$,m)C.

After running PerTuple, the removal of tuples are reflected in the domains of variables by projecting the altered constraints onto the domains of the variables in the constraints' scopes. PerTuple uses a data structure called an *index tree* that groups the equivalent tuples in a constraint relative to another constraint, implicitly exploiting the piecewise functionality of the two constraints. This data structure allows PerTuple's backtrack search to efficiently identify tuples that can be part of a valid solution to the combination.

Another algorithm used to enforce R($*$,m)C is AllSol [Karakashian, 2013]. Both PerTuple and AllSol operate in very similar manners, conducting searches over every connected subgraph of size m on the dual graph of a CSP. Where they differ is the manner in which they conduct their search. PerTuple focuses on a single relation R_i in a combination, conducting a search to find the first solution for each tuple or removing the tuple from R_i if a solution does not exist. This process needs to be repeated for every other relation in the combination as well, once all tuples in R_i have been processed.

AllSol, on the other hand, performs a single search over the combination, enumerating *all* solutions. Tuples which are not found in any of the enumerated solutions are removed from their respective relations to make the combination minimal. To improve the performance of AllSol, the search is stopped once all tuples have been marked as being a part of a solution. Further, while performing the search, if all tuples in the future subproblem *and* the tuples currently instantiated along the search path have already been marked, the subproblem is skipped.

Naively applying either AllSol or PerTuple to an entire CSP is cost prohibitive, even with small combination sizes. Instead, Karakashian *et al.* [2013] advocates to 1) weaken the consistency enforced by R($*$,m)C by creating combinations based on the minimal dual graph, resulting in the property wR($*$,m)C, and 2) localizing the scope of R($*$,m)C to the clusters formed by a tree decomposition of the CSP, resulting in the consistency properties cl-R($*$,m)C and cl-wR($*$,m)C. When using a tree decomposition, the size of m may be adjusted to the number of constraints in a particular tree cluster. This is denoted as cl-wR($*$,$\Psi(cl_i)$)C and ensures the minimality of each cluster in the decomposition. Karakashian [2013] has previously shown the advantage to using this approach, and we follow suit in this dissertation, enforcing cl-wR($*$,m)C exclusively by using a tree decomposition of the CSP.

5.2 PerFB Algorithm

This section introduces a new algorithm for enforcing R($*$,m)C called PerFB. At its core, the operation of PerFB is extremely similar to PerTuple, but dramatically improve its performance by using the data structures and partitions described in Chapter 4, and by identifying entire subproblems that have already been explored while performing searches over combinations of constraints.

5.2.1 Replacing the Index Tree with Coarse Blocks

Figure 5.3 provides an example of an index tree, which is, at its core, a *trie* (also known as a *prefix tree*). When conducting search over the relations in a combination, the trie can be walked with an instantiated tuple from another relation to identify the piecewise functional tuples supporting it. The index tree is conceptually similar to the data structures used to enforce PW-AC2 introduced in Chapter 4. Both serve to quickly identify a group of tuples in a relation that match a given signature. However, the index tree has a few notable downsides relative to the coarse block partitions.

The first disadvantage is the need to walk the tree each time a group of supporting tuples needs to be found. This operation is handled in constant time in PW-AC2 with only a modest amount of additional memory consumption. To find the supports for

R_1

	A	B	C
t_1	1	0	0
t_2	2	1	0
t_3	2	1	1
t_4	0	0	1
t_5	0	1	1
t_6	0	1	2
t_7	1	1	1

R_2

A	B	E
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1

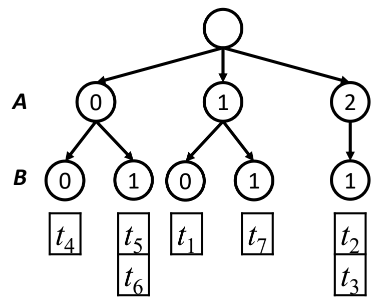


Figure 5.3: Two relations (R_1 and R_2) and the index tree corresponding to R_1 with respect to R_2.

an arbitrary τ from a relation R_i in some adjacent relation R_j using coarse blocks, we first identify $\mathit{Subscope}(R_i, R_j)$ which can be performed in constant time through the dual graph, and then use the table `blockLookup`$[R_j][\sigma_i][\tau]$ to immediately identify the group of supporting tuples for τ. Using the index tree, the subscope between the two relations is computed, then each value corresponding to the variables in the subscope are selected from the tuple and used to find the supporting group of tuples. However, this requires the index trees to have a universal canonical ordering of variables from root to leaf, which is fairly constricting, as the scopes of various relations may be arbitrarily ordered in the problem description.

The second disadvantage is the lack of additional metadata included at the leaves of the index tree, such as a counter indicating the number of living tuples in a particular group, or a pointer to other supporting groups in neighboring relations with the same signature.

Finally, the index trees are created for each pair of tuples. A straightforward but significant improvement would be to only construct index trees once per subscope, as is the case with the partitioning scheme (i.e., coarse blocks) used by PW-AC2.

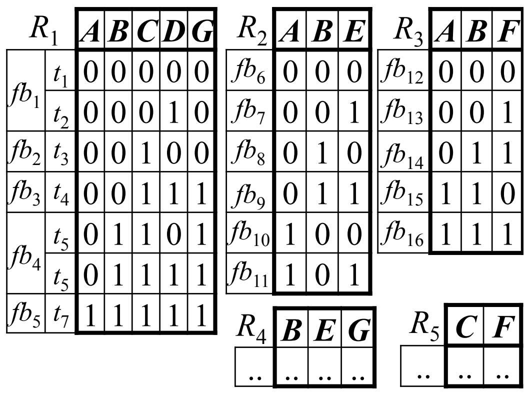
The lookahead mechanism for PerFB uses the block structures rather than index trees. This change was straightforward in STAMPEDE, as both the index tree and coarse blocks were written to the same interface, making coarse blocks a drop in replacement for the index tree. While no significant modification were needed to be made to PERTUPLE to use coarse blocks, the faster lookup times for supporting blocks play a role in the increased performance of PERFB.

5.2.2 Additional Equivalence Classes of Relations

In Chapter 4, we introduced the notion of a "coarse block" of tuples and exploited the equivalence classes induced, on the relation R_i of a constraint C_i, by C_i's neighbors in the dual graph. Here, we distinguish two additional types of such equivalence classes depending on the subset of neighbors considered: fine blocks (fb) and intermediate blocks (ib). Figures 5.4a and 5.4b illustrate these equivalence classes for R_1. The notations and data structures used in the following sections refer to this example.

Coarse blocks: Any single neighbor of C_i in the dual graph partitions R_i into a set of coarse blocks. In Figure 5.4a, $\mathit{subscope}(C_1{,}C_2) = \{A, B, C, D, G\} \cap \{A, B, E\} = \{A, B\} = o_1$. The tuples $t_{i\in[1,4]} \in R_1$ are equivalent for R_1 given $o_1{=}00$,[1] and consistent with (0,0,0) and (0,0,1) $\in R_2$. Indeed, $\pi_{o_1}(t_{i\in[1,4]} \in R_1) = (0, 0)$ and $\pi_{o_1}((0, 0, 0) \in R_2) = \pi_{o_1}((0, 0, 1) \in R_2) = (0, 0)$. Further, the above does not hold for any other tuple of R_1. Thus, $cb_1 = \{t_1, t_2, t_3, t_4\}$ is the coarse block of R_1 induced by $o_1{=}00$. The other two coarse blocks are $cb_2 = \{t_5, t_6\}$ and $cb_3 = \{t_7\}$. Similarly, $\mathit{subscope}(C_1, C_{j\in\{3,4,5\}})$ is $o_1 = \{A, B\}$, $o_2 = \{B, G\}$, and $o_3 = \{C\}$ respectively. Thus, o_1, o_2, and o_3 induce on R_1 the set of coarse blocks $\{cb_1, cb_2, cb_3\}$, $\{cb_4, cb_5, cb_6\}$, and $\{cb_7, cb_8\}$, respectively. Coarse blocks

[1]Abusing tuple/set assignment notation.



R_1		**A**	**B**	**C**	**D**	**G**
fb_1	t_1	0	0	0	0	0
	t_2	0	0	0	1	0
fb_2	t_3	0	0	1	0	0
fb_3	t_4	0	0	1	1	1
fb_4	t_5	0	1	1	0	1
	t_5	0	1	1	1	1
fb_5	t_7	1	1	1	1	1

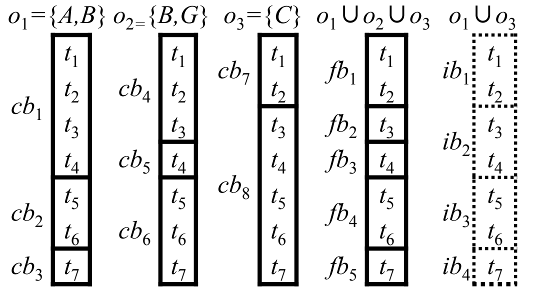
R_2	**A**	**B**	**E**
fb_6	0	0	0
fb_7	0	0	1
fb_8	0	1	0
fb_9	0	1	1
fb_{10}	1	0	0
fb_{11}	1	0	1

R_3	**A**	**B**	**F**
fb_{12}	0	0	0
fb_{13}	0	0	1
fb_{14}	0	1	1
fb_{15}	1	1	0
fb_{16}	1	1	1

R_4	**B**	**E**	**G**
..	..	..	..

R_5	**C**	**F**
..	..	..

(a) An example CSP.



(b) Coarse, fine, and intermediate blocks of R_i

Figure 5.4: An overview of different partitioning schemes.

are the partitions identified and exploited by Samaras and Stergiou [Samaras and Stergiou, 2005] and PW-AC2.

Fine blocks: When we consider all the constraints adjacent to C_i in the dual graph, they induce on R_i the finest possible partition, obtained by performing the intersections of all of R_i's coarse blocks. As a result, they yield the (unique) set of R_i's fine blocks. In Figure 5.4b, the set of fine blocks of R_1 is $\{fb_1, fb_2, \ldots, fb_5\}$. We note here that a fine block contains more than one tuple iff the scope of the constraint containing the tuple has at least one variable that *only* appears in the scope of that particular constraint.

Intermediate blocks: Finally, the partition induced on a relation R_i by a given combination of m constraints depends on the neighboring constraints of C_i that are included in m. The granularity of that partition is intermediate: not finer than R_i's fine partition and not coarser than any of its coarse partitions. For example, $\{C_2, C_5\} \subset neighbors(C_1)$ induce the intermediate blocks $\{ib_1, ib_2, ib_3, ib_4\}$.

In our prior work [Schneider *et al.*, 2014], the fine blocks found in the problem were computed and stored before preprocessing alongside the coarse blocks. However, while they may be of theoretical value, in practice we have found their use to be of limited practical import. In most problems, there is a one-to-one relationship between tuples and their fine blocks. There tend to be few relations whose constraint has a scope that includes a variable that was constrained only by that particular constraint. Further, the additional overhead to create the fine blocks during preprocessing and the extra complexity they induce on the algorithm makes them more hindrance than help.

5.2.3 The PerFB and AllSolFB Algorithms

Here, we describe FB-SearchSupport, which improves on PerTuple's SearchSupport [Karakashian *et al.*, 2010a]. The overall operation of PerFB is otherwise identical to PerTuple. Like PerTuple, PerFB takes as input a queue $\mathcal{Q}$ and Φ, where Φ is the set of all combinations of m relations. The queue is initialized to all the combination-relations pairs $\langle \varphi, R_i \rangle$ such that $\varphi \in \Phi$ and $R_i \in \varphi$. PerFB iterates over all tuples of a relation R_i in a combination φ, calling FB-SearchSupport to ensure that a tuple can be extended to a solution in the dual CSP induced by φ by conducting a backtrack search. In addition to the use of *static* coarse blocks (as opposed to the index tree), FB-SEARCHSUPPORT makes use of intermediate blocks, *dynamically* induced by the relations in φ.

PERFB improves PERTUPLE by reducing the cost of the search over the combination in SEARCHSUPPORT by exploiting dynamically induced intermediate blocks. The central idea of the improvement is to check, prior to the instantiation of a tuple, whether another tuple with the same signature has already had its subtree explored. If so, the result of the prior instantiation is returned, as the resulting subtree is guaranteed to be identical.

Algorithm 10: Modified methods of SEARCHSUPPORT for PERFB.

```
1  Method Label($C_i$: A constraint to assign):
2      $\tau \leftarrow$ AssignTuple()
3      node $\leftarrow$ ibTracker.getNode($C_i$,$\tau$)
4      if node.result $\neq$ $UNKNOWN$ then
5          return node.result
6      $consistent \leftarrow$ LookAhead()
7      if $consistent = false$ then
8          node.result $\leftarrow false$
9      else if $all\ constraints\ are\ assigned\ and\ consistent = true$ then
10         node.result $\leftarrow true$
11 Method Unlabel():
       ⋮
12     Update state as normal, removing changes caused by this node
       ⋮
13     ibTracker.resetNodes()
```

Algorithm 10 shows the modifications required to the search procedure used by PERTUPLE and ALLSOL. We refer to two functions, LABEL and UNLABEL, which are commonly used names when describing search over CSPs to refer to the process of instantiating and node in the tree and uninstantiating it, respectively [Prosser, 1993]. The modified search uses a new IBTracker, to determine if the current instantiation of

⟨constraint, tuple⟩ pair is equivalent to another pair that has already been searched. Algorithm 11 describes the operation of this class. `IBTracker` returns a node that contains one of three values: *unknown*, true, or false. If the node contains *unknown*, the current tuple is the first of its intermediate partition to be assigned at the current level of search. Otherwise, the boolean value stored in the result corresponds to whether the prior tuple from the same intermediate partition was consistent or not.

To accomplish this, class `IBTracker` effectively builds an index tree for the partition of tuples induced on the constraint at the current level of the subproblem search by the union of scopes it shares with the unassigned constraints. Each level of the trie corresponds to the intermediate block induced by the union of subscopes between the relation being assigned and the unassigned relations in the combination. When a tuple is instantiated during the search over a combination, it is added to the trie for that level.

Method LOOKUPNODE in Algorithm 11 returns a references to the leaf node of the trie, which can then be used to store the result of the current subproblem. Method CREATEINTERMEDIATEBLOCKSCOPE generates a bitset corresponding the union of variables in the subscopes between the current constraint C_i and its unassigned neighbors. Method GETNODE constructs a new path in the trie if the intermediate partition induced by the current tuple has not yet been evaluated, or the node corresponding to the previous result if it has. We assume that each node has a sparse map called `next` to provide pointers to child nodes in the trie. GETNODE walks through the variables in the bitset generated by Method CREATEINTERMEDIATEBLOCKS, generating new nodes when needed. Method RESETNODES clears out the generated trie and set of variables at a given level. An important implementation detail is the use of a pool of nodes that are recycled (and dynamically expanded when needed) to prevent frequent (de-)allocations from the construction and tear down of

Algorithm 11: Class IBTracker that tracks visited nodes in FB-SearchSupport.

```
/* Member variables of the tracker class */
rootNodes[$m$] // Root nodes of trie per level of subproblem search
ibScopes[$m$] // Subscope defining $ib$ per level of subproblem search
neighbors[$|\mathcal{C}|$] // Sparse set of global neighbors for each constraint
subscopes[$|\mathcal{C}|$,$|\mathcal{C}|$] // Stores bitset representation of a subscope
unassignedNodes // Sparse set of unassigned constraints in
    combination

Method resetNodes($level$):
    rootNodes[$level$].clear()
    ibScopes[$level$].clear()

Method createIntermediateBlockScope($C_i$):
    if size(unassignedNodes) $= 0$ then
        return $\emptyset$
    IBBitset $\leftarrow 0$
    for possNeigh $\in$ unassignedNodes $s.t.$ possNeigh $in$ neighbors[$C_i$] do
        IBBitset $|=$ subscopes[possNeigh, $C_i$]
    return IBBitset

Method getNode($\tau$, $level$):
    node $\leftarrow$ rootNodes[$level$]
    for varID $\in$ ibScopes[$level$] do
        val $\leftarrow$ $\tau$.atVar(varID)
        if val $\notin$ node.next() then
            node.next(val) $\leftarrow$ new node
            node.result $\leftarrow$ UNKNOWN
        node $\leftarrow$ node.next(val)
    return node

Method lookupNode(
        $C_i$: The instantiated constraint at $level$,
        $\tau$: The tuple assigned to $C_i$,
        $level$: The current level of the tuple search):
    if ibScopes[$level$] = then
        ibScopes[$level$] $\leftarrow$ createIntermediateBlockScope($C_i$)
    return getNode($tau$, $level$)
```

each trie.

Soundness of PerFB. The central conceit of PerFB is its ability to skip expanding nodes, thereby foregoing the processing of entire subproblems during the search over a combination. Node expansion is prevented iff the tuple assigned at the node has an identical signature to a previously explored node with respect to the remaining subproblem. Any variables not included in the signature cannot possibly have an impact on the remaining subproblem as they are, by definition, not constrained by any other constraints in the combination. The assignment of two tuples with identical signatures at *any* level of search will necessarily produce identical search trees. Therefore, PerFB will produce an identical result to PerTuple.

Note that the version of PerFB implemented in Stampede forgoes the explicit creation of fine blocks entirely. The mechanism used to identify tuples with equivalent signatures during a subproblem search is extremely efficient, and any tuples that would have been in a fine block will be implicitly identified as part of the same intermediate block during the subproblem search, negating the benefit from pre-computing them.

Complexity. When deleting a tuple during search, it is important to maintain the correct counts of living tuples remaining in coarse blocks. Each tuple deletion requires $O(e^2)$ updates in the worst case. Updates are performed in constant time thanks to the `blockLookup` table. The cost of these updates is, in practice, greatly dwarfed by that of FB-SearchSupport. The time complexity of PerFB is identical to that of PerTuple, and dominated by the $O(t^{m-1})$ search conducted in FB-SearchSupport [Karakashian *et al.*, 2010a]. Additionally, PerFB performs at most as much work in (FB)-SearchSupport as PerTuple does, because

$$\bigcup_{R_j \in \varphi \setminus \{R_i\}} \mathit{subscope}(R_i, R_j) \tag{5.1}$$

is the same as $\mathit{scope}(R_i)$ in the worst case. At each level of search in FB-SearchSupport, the trie is composed of $O(t)$ tuples. Thus, an additional $O(m \cdot t)$ space is required for PerFB to store the results of previously searched subtrees at each level of search in FB-SearchSupport.

The modifications to FB-SearchSupport extend naturally and easily to AllSol, producing the algorithm AllSolFB. Both AllSol and PerTuple use the same SearchSupport procedure in Stampede, so any improvements made to SearchSupport carry directly into AllSol. This is unique to Stampede's implementation of PerTuple and AllSol, making this work the first time AllSolFB has ever been evaluated.

5.3 Empirical Evaluation

In this section we evaluate the performance of algorithms PerFB, PerTuple, AllSolFB, and AllSol. Each algorithm configuration uses the tree decomposition to direct propagation, using $m = 3$ and $m = \Psi$ to enforce cl-wR($*$,m)C and cl-wR($*$,$\Psi(cl_i)$)C, respectively. We label each algorithm with the level of consistency enforced (e.g., PerFB^{m3} or PerFB$^{\psi}$). The experiments were run on 107 binary and 72 non-binary CSPs (a total of 4,970 instances). These benchmarks were selected from the CPAI08 dataset[2]. We excluded benchmarks which were unable to be completed by any algorithm presented in this dissertation.

PerTuple and PerFB can optionally store the solutions found for each tuple when enforcing R($*$,m)C to prevent repeated searches for the same solution on sub-

[2]http://www.cril.univ-artois.fr/CPAI08/

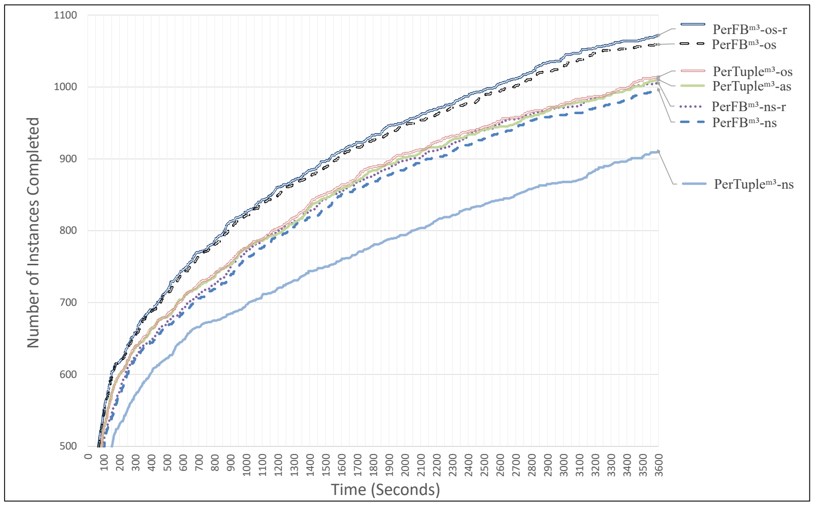
sequent calls to the algorithm. We tested configurations of PerTuple and PerFB that store one solution per tuple (-*os*) and no solutions (-*ns*). We also tested PerTuple with a support scheme that stores a list of all valid solutions that a tuple participates in (-*as*). We have previously found this to be counter productive in the context of PerFB due to the memory overhead required to store all tuples, with minimal advantage due to the overlapping functionality of storing supports and computing the intermediate blocks in PerFB. We also examined two variations of PerFB and AllSolFB which only compute the intermediate blocks at the root node, rather than at every node in search (-*r*).

Search over each CSP was performed using Stampede to find the first consistent solution, using the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. The solver was limited to one hour and 8GB of memory. When an algorithm timed out or ran out of memory (OOM), it is recorded as having taken the full hour. It is important to note that *all* of the tested algorithms in this section explore identical search trees, so long as the same ordering heuristics are used.

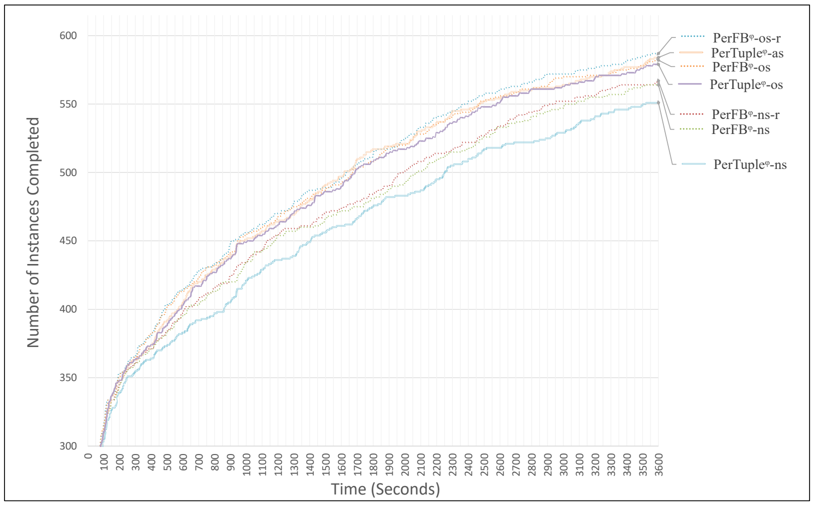
5.3.1 Binary Benchmarks

We begin by examining binary instances.

Figure 5.5 provides a cumulative chart showing the number of instances solved within a given time frame using PerFB and PerTuple on binary benchmarks. The performance of PerFB is considerably higher than PerTuple for $m = 3$, and boasts a modest improvement for $m = \Psi$. The version of PerFB which *only* checks for redundant intermediate blocks at the root of the search tree outperforms the full version for both tested combination sizes. This is not particularly surprising. One of the main benefits of checking for redundant subproblems is finding subtrees which are



(a) $m = 3$



(b) $m = \Psi$

Figure 5.5: Cumulative charts showing number of binary instances completed in N seconds for PerFB and PerTuple.

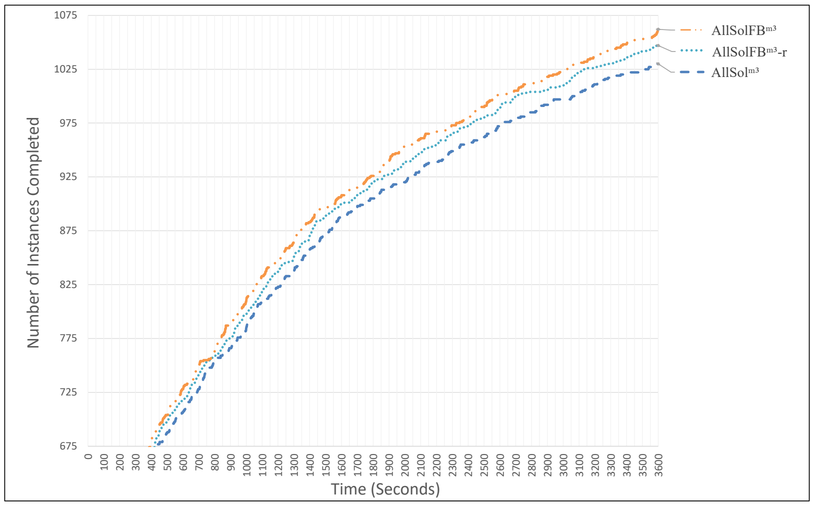
already known to be *consistent*, but if a subtree is consistent, PerFB and PerTuple will have already found the first solution and returned from SearchSupport.

Using PerFB on subproblems the size of clusters is somewhat pathological for PerFB, especially with variants that only check for redundancy at the root of the search tree. The size of the intermediate blocks (and thus, the number of tuples that could potentially be skipped) is likely smaller when m is large; the root node is likely to share many of the variables in its scope with unassigned constraints. Should it share *all* of its scope with the unassigned constraints in the subproblem, all intermediate blocks will have a single tuple. In binary instances, this is exceedingly likely as the cardinality of all scopes in the problem is two.

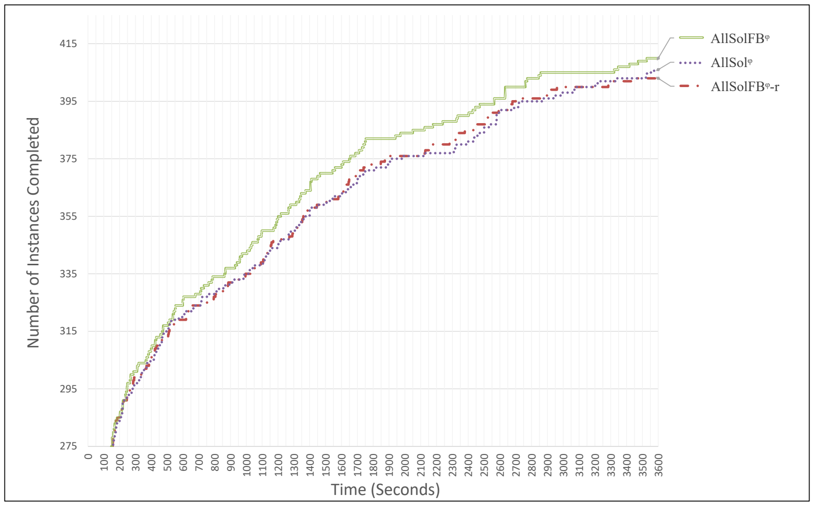
Figure 5.6 provides a cumulative chart showing the number of instances solved within a given time frame using AllSolFB and AllSol on binary benchmarks. Unlike PerFB, here we can clearly see the value of checking for redundant subproblems during search, due to AllSol exploring the entire search tree until all tuples are marked or all solutions in the subproblem are found. At least one variant of both AllSolFB and PerFB dominate AllSol and PerTuple (respectively) on binary problems.

Figure 5.7 shows scatter charts comparing the time required to complete each instance using the variants of PerTuple, PerFB, AllSol, and AllSolFB which were able to solve the most instances in the provided hour. Points below the diagonal are instances that were completed faster by PerFB or AllSolFB. Notably there are a few instances when processing combinations of size Ψ that run out of memory with PerFB and AllSolFB, but successfully complete with PerTuple and AllSol (even when storing all solutions with PerTuple).

This is likely due to the increased memory required to create the coarse block structures used for lookahead in PerFB and AllSolFB relative to index trees. However,

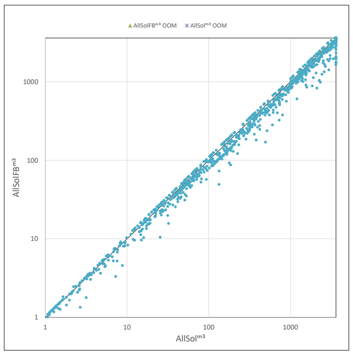


(a) $m = 3$

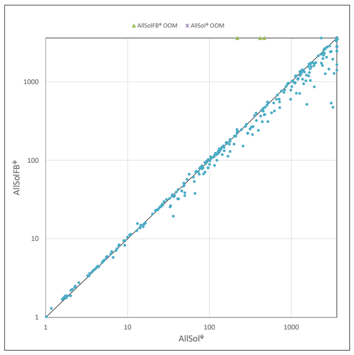


(b) $m = \Psi$

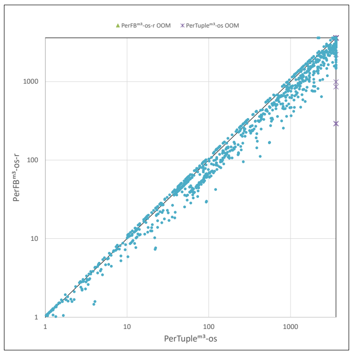
Figure 5.6: Cumulative charts showing number of binary instances completed in N seconds for AllSolFB and AllSol.



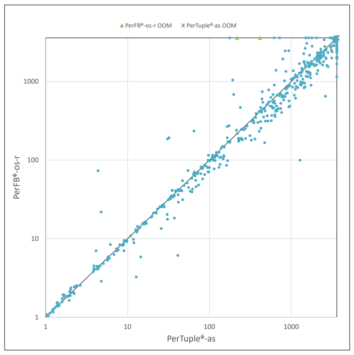
(a) AllSolFB^{m3} v. AllSol^{m3}



(b) AllSolFB$^{\Psi}$ v. AllSol$^{\Psi}$



(c) PerFB^{m3}-os-r v. PerTuple^{m3}-os



(d) PerFB$^{\Psi}$-os-r v. PerTuple$^{\Psi}$-as

Figure 5.7: Scatter charts comparing relative performance of PerTuple and PerFB (top) and AllSol and AllSolFB (bottom) on binary benchmarks. Note the logarithmic scale.

the coarse block structures play a significant role in the gains in performance that we see with PerFB when $m = \Psi$; as mentioned above, detecting redundant tuples is unlikely. The exception to this are random problems, which tend to have many tuples and whose lack of structure can produce situations conducive to PerFB. Table 5.1 provides statistics for the number of supports found, subproblems skipped because of the intermediate blocks at both the root node and subproblem, and the total number of nodes visited during searches over combinations. Clearly, the effort spent trying to detect redundant subproblems with PerFB is wasted, *especially* on small combinations. Note that these statistics were collected across all problems, so the values from one algorithm to another are incomparable (some algorithms completed many

Table 5.1: Various statistics for the combination searches on binary problems.

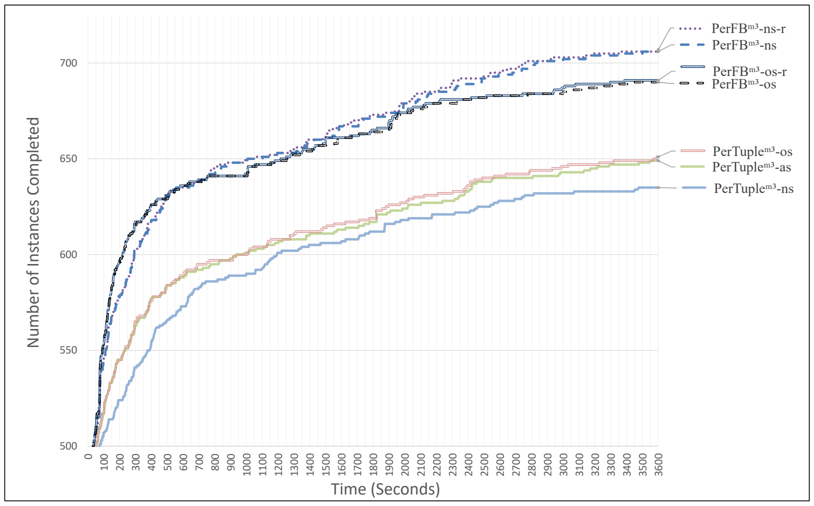
	Avg. # Support Skips	StdDev # Support Skips	Avg. # IB Skipped (root)	StdDev # IB Skipped (root)	Avg # IB Skipped (ss)	StdDev # IB Skipped (ss)	Avg # Node Visits	StdDev # Node Visits
AllSolFB$^{\Psi}$-r	-	-	833,249	3,118,043	-	-	260,785,995	622,556,014
AllSolFB$^{\Psi}$	-	-	926,806	3,343,735	8,387,554	29,964,280	232,192,378	585,008,180
AllSolFB^{m3}-r	-	-	20,887,207	35,196,560	-	-	755,073,820	1,063,734,923
AllSolFB^{m3}	-	-	21,266,136	35,863,484	35,380,901	60,804,353	718,161,884	1,041,962,992
AllSol$^{\Psi}$	-	-	-	-	-	-	651,190,657	716,074,954
AllSol^{m3}	-	-	-	-	-	-	821,231,523	878,860,367
PerFB$^{\Psi}$-ns-r	-	-	7,411,033	21,355,873	-	-	180,549,404	372,407,938
PerFB$^{\Psi}$-ns	-	-	7,489,158	21,843,839	205,834	900,826	169,989,629	363,172,224
PerFB$^{\Psi}$-os-r	10,789,232	27,824,459	5,651,021	16,245,831	-	-	196,711,535	439,540,671
PerFB$^{\Psi}$-os	10,122,982	26,442,130	5,223,605	15,007,916	272,828	1,568,792	169,704,833	358,547,313
PerFB^{m3}-ns-r	-	-	112,207,533	200,753,924	-	-	316,840,792	551,278,442
PerFB^{m3}-ns	-	-	109,279,829	195,469,209	315	812	310,186,881	541,571,633
PerFB^{m3}-os-r	169,614,137	299,036,500	57,412,913	103,092,200	-	-	141,477,716	241,551,663
PerFB^{m3}-os	161,575,084	285,863,116	54,563,821	97,872,297	398	975	135,990,097	234,948,966
PerTuple$^{\Psi}$-as	11,152,170	29,676,837	-	-	-	-	151,550,180	159,908,292
PerTuple$^{\Psi}$-ns	-	-	-	-	-	-	194,021,065	204,573,996
PerTuple$^{\Psi}$-os	9,626,786	26,032,042	-	-	-	-	156,802,512	165,808,837
PerTuple^{m3}-as	193,493,822	349,807,123	-	-	-	-	95,879,010	87,683,229
PerTuple^{m3}-ns	-	-	-	-	-	-	142,299,129	130,867,614
PerTuple^{m3}-os	191,442,904	342,831,195	-	-	-	-	94,210,046	86,702,144

more instances than others, which would skew the values).

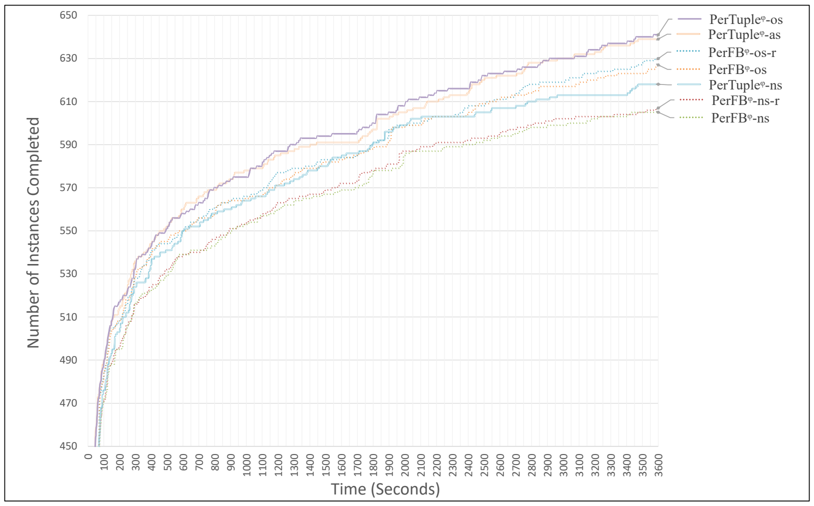
Finally, Table 5.2 provides the results of a paired t-test between all algorithms evaluated in this chapter on binary problems. PerFB^{m3}-os-r and AllSolFB^{m3} both show statistical significance with respect to all other tested algorithms.

5.3.2 Non-Binary Benchmarks

The results for non-binary benchmarks contain a few notable differences from the binary benchmarks. Figure 5.8 provides a cumulative chart showing the number of instances solved within a given time frame using PerFB and PerTuple on non-binary benchmarks. The disadvantage of PerFB relative to PerTuple on large combination sizes is is emphasized here, as PerTuple outperforms both versions of PerFB. However, on smaller combination sizes, PerFB still leads the pack.



(a) $m = 3$



(b) $m = \Psi$

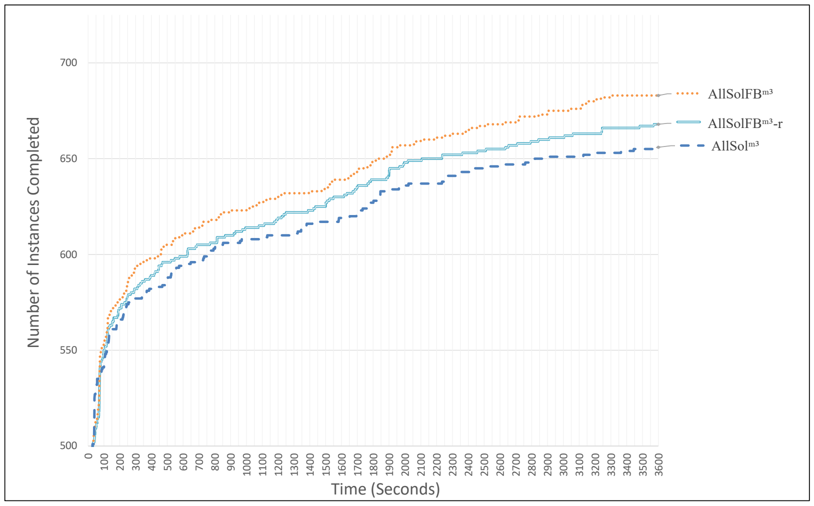
Figure 5.8: Cumulative charts showing number of non-binary instances completed in N seconds for PERFB and PERTUPLE.

Table 5.2: Pairwise t-test results for m-wise consistency algorithms with the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic on binary problems. Cells are labeled "T" if the results were significantly different (p <.05) and favored the row entry.

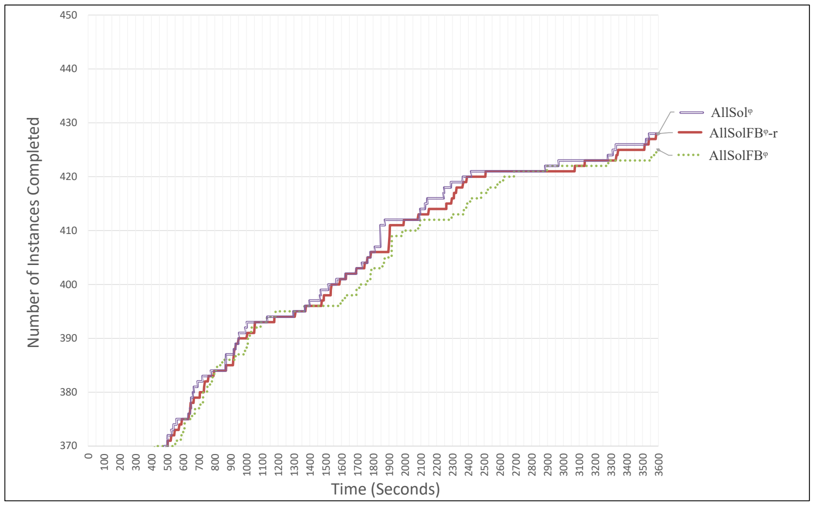
	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$-os	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}-os
AllSolFB$^{\Psi}$-r	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSolFB$^{\Psi}$	T	-	-	-	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSolFB^{m3}-r	T	T	-	-	T	T	T	T	T	T	T	T	-	-	T	T	T	T	T	T
AllSolFB^{m3}	T	T	T	-	T	T	T	T	T	T	T	T	-	-	T	T	T	T	T	T
AllSol$^{\Psi}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSol^{m3}	T	T	-	-	T	-	T	T	T	T	T	T	-	-	T	T	T	T	T	T
PerFB$^{\Psi}$-ns-r	T	T	-	-	T	-	-	T	-	-	-	-	-	-	-	T	-	-	-	-
PerFB$^{\Psi}$-ns	T	T	-	-	T	-	-	-	-	-	-	-	-	-	-	T	-	-	-	-
PerFB$^{\Psi}$-os-r	T	T	-	-	T	-	T	T	-	T	-	-	-	-	-	T	T	-	-	-
PerFB$^{\Psi}$-os	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	T	-	-	-	-
PerFB^{m3}-ns-r	T	T	-	-	T	-	T	T	T	T	-	T	-	-	T	T	T	-	T	-
PerFB^{m3}-ns	T	T	-	-	T	-	T	T	T	T	-	-	-	-	T	T	T	-	T	-
PerFB^{m3}-os-r	T	T	T	-	T	T	T	T	T	T	T	T	-	T	T	T	T	T	T	T
PerFB^{m3}-os	T	T	T	-	T	T	T	T	T	T	T	T	-	-	T	T	T	T	T	T
PerTuple$^{\Psi}$-as	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	T	T	-	-	-
PerTuple$^{\Psi}$-ns	T	T	-	-	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PerTuple$^{\Psi}$-os	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	T	-	-	-	-
PerTuple^{m3}-as	T	T	-	-	T	-	T	T	T	T	-	T	-	-	T	T	T	-	T	-
PerTuple^{m3}-ns	T	T	-	-	T	-	T	T	T	T	-	-	-	-	T	T	T	-	-	-
PerTuple^{m3}-os	T	T	-	-	T	-	T	T	T	T	-	T	-	-	T	T	T	T	T	-

Figure 5.9 provides the cumulative chart for AllSolFB and AllSol on non-binary benchmarks. Here, the difference in performance when attempting to identify redundant subproblems is even more pronounced than it was with PerFB. The performance of AllSol$^{\Psi}$ is significantly better than that of AllSolFB$^{\Psi}$. Attempting to identify redundant subproblems at the root only in AllSolFB$^{\Psi}$-r does not impact the performance of the algorithm greatly, which is not surprising. The number of times the root node is evaluated in AllSol with $m = \Psi$ is very low, as each triggering of the propagator will ensure minimality over the entire cluster, and the root node is only instantiated once per tuple in its relation.

This rationale is further validated by the scatter plots in Figure 5.10, which show



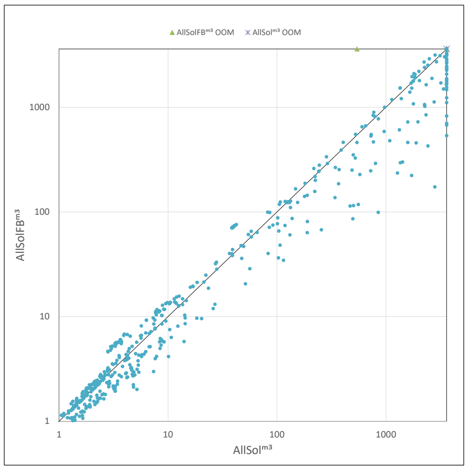
(a) $m = 3$



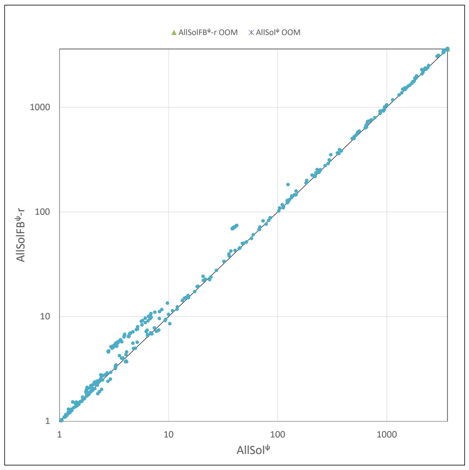
(b) $m = \Psi$

Figure 5.9: Cumulative charts showing number of binary instances completed in N seconds for AllSolFB and AllSol.

nearly identical performance between the AllSol$^{\Psi}$ and AllSolFB$^{\Psi}$-r, but significantly improved performance for AllSolFB^{m3}, with a gap that appears to widen as the problems become more difficult.

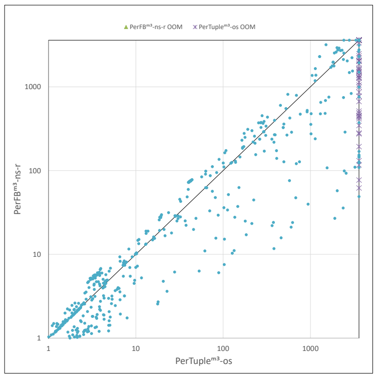


(a) $m = 3$

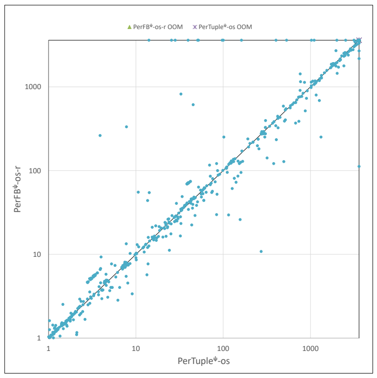


(b) $m = \Psi$

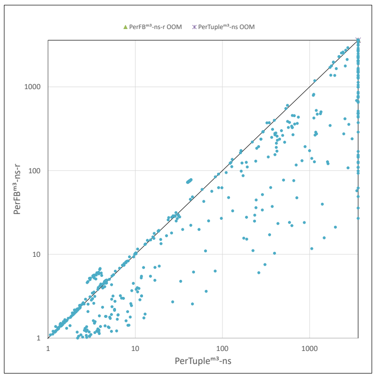
Figure 5.10: Scatter charts comparing relative performance of AllSol and AllSolFB on non-binary benchmarks. Points below the diagonal favor AllSolFB.



(a) $m = 3$



(b) $m = \Psi$



(c) $m = 3$ w/out supports

Figure 5.11: Scatter charts comparing relative performance of PerTuple and PerFB on non-binary benchmarks. Points below the diagonal favor PerFB.

The scatter plots in Figure 5.11 show the benefit of detecting intermediate blocks when enforcing R($*$,m)C. Figure 5.11a compares the two algorithms that solved the most non-binary instances, PerFB^{m3}-ns-r and PerTuple^{m3}-os. Tracking solutions in these problems is extremely cost prohibitive, as made evident by the large cluster of instances on the right side of the chart which PerFB^{m3}-ns-r finished but PerTuple^{m3}-os was unable to due to memory limits. Removing the support structure from PerTuple^{m3}-os results in the chart in Figure 5.11c, where it becomes clear that without the (at times) memory-intensive support tracking, PerTuple^{m3} is unable to keep with PerFB^{m3}-ns-r. The chart for PerFB$^{\Psi}$ supports the previous assertion that, with a few exceptions, the use of intermediate blocks at large combination sizes is unlikely to be beneficial.

Table 5.3: Pairwise t-test results for m-wise consistency algorithms with the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic on non-binary problems. Cells are labeled "T" if the results were significantly different (p <.05) and favored the row entry.

	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$-os	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}-os
AllSolFB$^{\Psi}$-r	-	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSolFB$^{\Psi}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSolFB^{m3}-r	T	T	-	-	T	T	T	T	T	T	-	-	-	-	T	T	T	T	T	T
AllSolFB^{m3}	T	T	T	-	T	T	T	T	T	T	-	-	-	-	T	T	T	T	T	T
AllSol$^{\Psi}$	T	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSol^{m3}	T	T	-	-	T	-	T	T	T	T	-	-	-	-	T	T	T	T	T	T
PerFB$^{\Psi}$-ns-r	T	T	-	-	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PerFB$^{\Psi}$-ns	T	T	-	-	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
PerFB$^{\Psi}$-os-r	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	-	-	-	-	-
PerFB$^{\Psi}$-os	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	-	-	-	-	-
PerFB^{m3}-ns-r	T	T	T	T	T	T	T	T	T	T	-	T	-	-	T	T	T	T	T	T
PerFB^{m3}-ns	T	T	T	T	T	T	T	T	T	T	-	-	-	-	T	T	T	T	T	T
PerFB^{m3}-os-r	T	T	T	T	T	T	T	T	T	T	-	-	-	T	T	T	T	T	T	T
PerFB^{m3}-os	T	T	T	T	T	T	T	T	T	T	-	-	-	-	T	T	T	T	T	T
PerTuple$^{\Psi}$-as	T	T	-	-	T	-	T	T	T	T	-	-	-	-	-	T	-	-	-	-
PerTuple$^{\Psi}$-ns	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	-	-	-	-	-
PerTuple$^{\Psi}$-os	T	T	-	-	T	-	T	T	T	T	-	-	-	-	-	T	-	-	-	-
PerTuple^{m3}-as	T	T	-	-	T	-	T	T	T	T	-	-	-	-	-	T	-	-	T	-
PerTuple^{m3}-ns	T	T	-	-	T	-	T	T	-	-	-	-	-	-	-	T	-	-	-	-
PerTuple^{m3}-os	T	T	-	-	T	-	T	T	T	T	-	-	-	-	T	T	T	T	T	-

We conclude with the paired t-tests for non-binary instances, which firmly establishes the weakness of this approach using large combination sizes on non-binary problems, as the original versions of PerTuple and AllSol are more effective there. However, the use-cases for establishing minimality over subproblems of those sizes are somewhat limited, as the applying minimality over smaller combination sizes tends to be more effective overall.

Summary and Future Work

In this chapter, we introduced two new algorithms for enforcing R($*$,m)C: PerFB and AllSolFB. We empirically evaluated both algorithms and showed their dominance relative to PerTuple and AllSol when using small combination sizes, and evaluated the cause of their deficiencies on large combinations. Further improvements could be made to both algorithms by incorporating lessons learned from PW-CT into the subproblem search. Operating on bitsets has been show to be extremely advantageous in the context of tabular reduction, and could likely be extended to enforce R($*$,m)C.

Chapter 6

Dangle Identification: Dynamically Identifying and Solving Tractable Branches of Search

Although the cost of solving general CSPs is exponential in the number of variables in the problem, specific CSPs may be tractable under certain conditions. Tree-structured CSPs, for example, are tractable, but rare in practice, and can solved in a backtrack-free manner after enforcing directional arc consistency [Dechter and Pearl, 1988; Freuder, 1982a]. As mentioned in Section 2.3.3, prior work that builds on the tractability of tree-structured CSPs includes identifying a cycle-cutset that induces a tree-structured CSP and using the induced width of an ordering of the variables in order to bind the computational cost of search.

The importance of acyclic graphical models extends well beyond CSPs and is an important property in databases [Beeri *et al.*, 1983; Maier, 1983] and optimization problems [Derenievicz and Silva, 2018]. Prior approaches for exploiting a tree structure have largely focused on either inducing acyclicity (as in the cycle-cutset method [Bidyuk and Dechter, 2004; Dechter and Pearl, 1987]) or binding the search effort by the induced width of an ordering of the variables (as in adaptive consistency [Dechter and Pearl, 1988; Dechter, 1996]) or the treewidth of a tree decomposition of the constraint network (as in the BTD method [Jégou and Terrioux, 2003b]).

In this chapter, we advocate to monitor the structure of the problem as the search proceeds, variables are instantiated, and the constraint network becomes increasingly sparse. Our goal is to detect tractable subproblems as they dynamically unfold during search. Such subproblems appear as *dangling* off the CSP being solved. We propose to dynamically detect these dangles during search and remove them after determining their solvability so as to reduce the size of the search tree. To detect acyclic subproblems, we use the Graham Reduction (GYO) [Graham, 1979; Yu and Ozsoyoglu, 1979], which operates on the hypergraph of a CSP. The solvability of these subproblems is determined in polynomial time using either directional pairwise consistency before allowing search to proceed.

6.1 Background and Related Work

Acyclicity is an important property of a CSP (and of a database schema). Tree-structured binary CSPs can be solved in a backtrack-free manner after enforcing directional arc consistency [Dechter and Pearl, 1988; Freuder, 1982a]. Beeri *et al.* [1983] showed that, restated for non-binary CSPs, the following conditions (among others) are equivalent: the hypergraph is α-acyclic, the GYO Reduction succeeds on the hypergraph, pairwise consistency guarantees global consistency, and the CSP has a join tree.

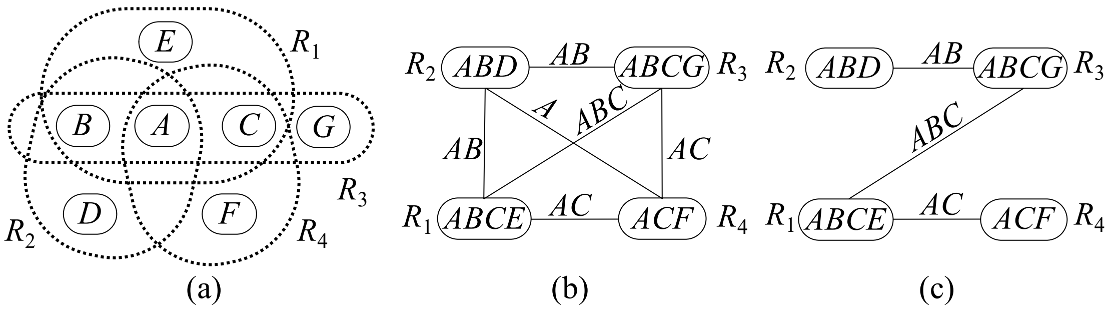


Figure 6.1: Graphical representations of a simple CSP: (a) the hypergraph, (b) the dual graph, and (c) a minimal dual graph.

The GYO Reduction is used to determine whether or not a database schema is acyclic [Graham, 1979; Yu and Ozsoyoglu, 1979]. In the context of a CSP, it repeatedly removes from the hypergraph all vertices (i.e., CSP variables) that are contained in at most one hyperedge. This operation ends either when all vertices are removed (which occurs when the hypergraph is acyclic), or when every vertex has a degree of two or more. Given a hypergraph $H = (V, E)$ where V are the vertices and E are the hyperedges, the two rules of the GYO Reduction are: (1) *Hyperedge removal*: If two hyperedges $e, f \in E$ are such that e is properly contained in f, remove e from E. (2) *Node removal*: If a vertex $v \in V$ is contained in at most one hyperedge in E, remove v from V and also from the hyperedge where it appears. For the hypergraph in Fig. 6.1, the node-removal rule can remove the variable D from R_2. Then, the hyperedge-removal rule can remove R_2 because it is properly contained in R_1.

Woodward *et al.* [2011b] propose an algorithm for enforcing relational neighborhood inverse consistency by running a backtrack search on the dual graph induced by a constraint and its neighbors. They improve the performance of this search on the dual graph by dynamically identifying dangles. Because the dangles are iteratively identified on (and removed from) the dual graph as vertices of degree 1, they are only a subset of the dangles identified on the hypergraph by the GYO Reduction.

The Cycle-Cutset method of Dechter and Pearl [1987] identifies a set of variables in a binary CSP whose removal transforms the constraint network into a tree. Using backtrack search, we find a solution to the cutset nodes then try to expand this solution to the tree-structured subproblem in a backtrack-free manner. In case of failure, we backtrack to the cutset nodes. Dechter and Pearl also propose to check during search whether the remaining CSP is tree structured.

The closest prior work to ours is that presented by Sabin and Freuder [1997]. They present a technique for identifying dangles on binary CSPs using a modified version of the (at the time state of the art) MAC algorithm. They dub the modified algorithm MACE, which operates by first ensuring the problem is fully arc-consistent, then removing variables which do not participate in any cycles from the problem. As new variables are instantiated, variables which then become cycle-free are separated and prevented from being instantiated until all variables participating in cycles are assigned, at which point the separated variables can be extended into a full solution backtrack-free.

However, the method presented by Sabin and Freuder [1997] has three major downsides. The first is that the consistency algorithm used is built-in to their technique, preventing the use of other (or dynamically selected) propagation techniques. The second downside is its reliance on non-standard variable ordering heuristics to drive the problem towards a cutset. While this technique may be beneficial in some problems, the efficacy of modern variable ordering heuristics like $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ are undeniable. Their results also do not evaluate the overhead required to identify the cutset and remove the free variables, but instead base their results on the number of constraint checks (a common measure at the of the paper's publication), though this overhead is likely to be small on binary problem. The third downside is it is not trivially extendable to non-binary problems.

Other work close in spirit to the work presented in this chapter is that which interleaves search and inference by variable elimination (which corresponds to adaptive consistency [Dechter and Pearl, 1988]) as proposed by Rish and Dechter [2000] for SAT and by Larrosa [2000] for binary CSPs. Variable elimination on subproblems of bounded width k is done by applying the resolution rule on the clause of a SAT theory when it does not yield more than k resolvents [Rish and Dechter, 2000] and by replacing a variable of degree k by a constraint of arity k over the neighbors of the variable of a binary CSP [Larrosa, 2000]. Both approaches add new clauses/constraints to the problem. Our approach does not require the addition of any new constraints to the problem but can be seen as a generalization of algorithm VarElimSearch [Larrosa, 2000] to non-binary CSPs with $k = 1$. However, in order to achieve this generalization, one needs to *add* a constraint normalization step, which corresponds to the hyperedge-removal rule of the GYO Reduction in Larossa's procedure VarElim.

6.2 Dangle Identification

In this section, we discuss the use of the hyper and dual graphs in combination with GYO. We show how to efficiently implement the hyperedge-removal rule of GYO, and provide an algorithm for identifying and removing dangles from a subproblem during search.

6.2.1 Which Graph to Use

We advocate (1) identifying dangles by running the GYO Reduction on the hypergraph of a CSP and (2) using the dual graph of the CSP to determine the subscopes incident to a given CSP constraint (which is needed in the edge-removal rule of the GYO Reduction). In this section, we caution against errors that may occur in this context.

Redundancy removal may cause errors in search. It seems natural to use a minimal dual graph of the CSP to link the subscopes to their incident constraints because a minimal dual graph is typically sparser than the full dual graph. We show, with an example, that using a minimal dual graph can yield unsound results during search. Consider the dual graph shown in Figure 6.2, a sub graph of the dual graph of the famous Zebra puzzle.[1]

The edges represented with dotted lines are redundant and can be removed. Assume the search instantiates the variable K. The hyperedge-removal rule of the GYO

[1]https://en.wikipedia.org/wiki/Zebra_Puzzle

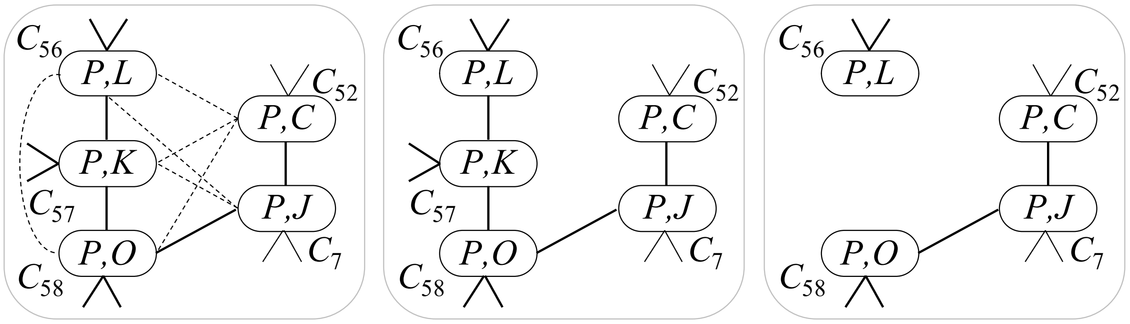
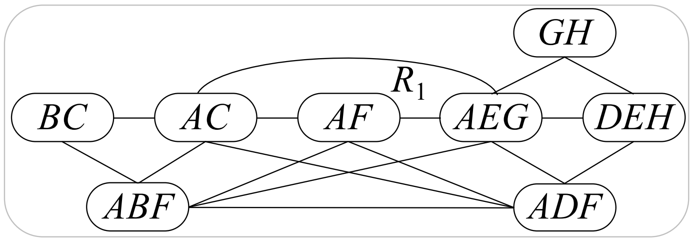


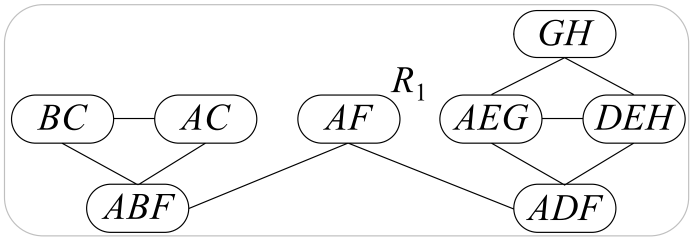
Figure 6.2: A subgraph of the dual graph of the Zebra puzzle with redundant edges removed, variable k instantiated, and the subsumed constraint C_{57} removed.

reduction removes C_{57} because it is properly contained in C_{56} (and C_{58}). As a result, C_{56} is disconnected from the other constraints where the variable P appears. This situation may cause errors because we lose the connectedness property and the two components may be incorrectly processed in isolation from one another. For example, we could miss 'grabbing' all the relevant constraints to include in a dangle. Directional pairwise consistency on this disconnected 'subdangle' may assume solvability whereas the 'full' dangle may not contain a solution. Generally speaking, to avoid this issue, one would have to reconnect all the vertices of the dual graph to which the subscope (potentially more than one) of the removed constraint is incident. Because some variables in the subscope may have been instantiated during search or unassigned by backtracking, the situation may become further complicated. Thus, we believe it non-trivial to use the minimal dual graph for the purposes of identifying neighborhoods for use in the GYO Reduction.

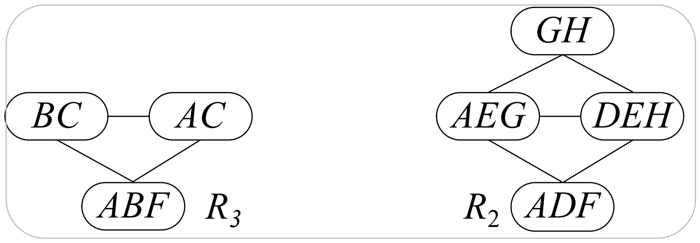
Redundancy removal may break the connectedness property. Even without the contribution of search, we may lose the connectedness property if we use a minimal dual graph. Fig. 6.3 shows the dual graph of a simple CSP, then a minimal dual graph obtained by redundancy removal, and the application of an edge-removal step of the GYO Reduction. R_1, which is connected to R_2 and R_3 in the displayed minimal dual graph, is contained in both. Its removal by the edge-removal step of the GYO Reduction, yields the disconnected minimal dual graph of Figure 6.3c.



(a) The dual graph of a CSP.



(b) A minimal dual graph of Fig. 6.3a.



(c) GYO Reduction on graph in Fig. 6.3b.

Figure 6.3: The dual graph of a CSP, followed by redundancy removal, then an edge removal.

Acyclicity is not easy to detect on the dual graph. For now, we choose to not use a minimal dual graph in order to avoid pitfalls like the ones identified above. The next question is: can all the dangles in the hypergraph be identified starting from vertices of degree 1 in the 'full' dual graph (i.e., the complete intersection graph)? Unfortunately, no, this is not possible. Figure 6.1 (b) shows a dual graph where no vertex has degree 1, yet the CSP is acyclic. Thus, checking the degree of vertices in the dual graph is not sufficient.

What to use? For the above reasons, we use the GYO Reduction (operating on the hypergraph) to detect acyclic subproblems and the dual graph to identify the subscopes incident to a given constraint. Because we use the 'full' dual graph, the degree of a vertex in the dual graph may be large. Consequently, we need to find a quick way to determine whether or not one constraint is subsumed by another.

6.2.2 Fast Hyperedge Removal

The hyperedge-removal rule of the GYO Reduction requires identifying, for a given constraint c_i, all the constraints c_j such that $scope(c_i) \subseteq scope(c_j)$. Importantly, this operation must be done dynamically during search, where an instantiated variable must be ignored in the scope of all the constraints. A naive implementation would perform a subset check on the scopes of the constraints, which requires $O(n \cdot log(n))$ checks for a sorted implementation or a $O(n)$ for a hashtable-based implementation , where n is the size of a constraint's scope. Instead of doing this subset operation, we propose to compare the cardinality of the $scope(c_i)$ (i.e., $|scope(c_i)|$) and the cardinality of each subscope incident to c_i in the dual graph. Let $subscopes(c_i)$ denote the *set* of subscopes incident to constraint c_i in the dual graph.

Proposition 5. *A constraint $c_i \in \mathcal{C}$ has a scope which is a subset of the scope of another constraint iff $\exists\, \sigma \in subscopes(c_i)$ such that $|scope(c_i)| = |\sigma|$, where $|scope(c_i)|$ and $|\sigma|$ do not count any removed or instantiated variables.*

Proof. ($\Rightarrow$) If the constraint c_i is properly contained in another constraint, then there exists $c_j \in \mathcal{C}$ such that $scope(c_i) \cap scope(c_j) = scope(c_i)$. If we denote $\sigma_{ij} = scope(c_i) \cap$

$scope(c_j)$, then $scope(c_i) = \sigma_{ij}$, which implies $|scope(c_i)| = |\sigma_{ij}|$. In addition $\sigma_{ij} \in subscopes(c_i)$ by definition of $subscopes(c_i)$.

($\Leftarrow$) Conversely, assume that $\exists\, \sigma \in subscopes(c_i)$ such that $|scope(c_i)| = |\sigma|$. Given $\sigma \in subscopes(c_i)$, then there exists a constraint $c_j \in \mathcal{C}$ such that $\sigma = scope(c_i) \cap scope(c_j)$. Thus, $\sigma \subseteq scope(c_i)$ and $\sigma \subseteq scope(c_j)$. Given that $\sigma \subseteq scope(c_i)$ and $|\sigma| = |scope(c_i)|$, then $\sigma = scope(c_i)$. We have $\sigma \subseteq scope(c_j)$ and $\sigma = scope(c_i)$, then $scope(c_i) \subseteq scope(c_j)$. Thus, c_i is properly contained in another constraint. $\square$

Our dangle identification algorithm, presented below, exploits Proposition 5.

6.3 Dangle Identification Algorithm

Below we provide an iterative version of the GYO Reduction that exploits Proposition 5 for the hyperedge-removal step. Our pseudocode makes use of the following global variables: queue is the set of constraints that need to be checked whether or not they are leaves and activeVars and activeCons are the set of variables and constraints, respectively, that have not yet been removed from the problem. Our algorithm also relies on several counters and sets that are accessed and updated through the following functions. The function incidentSubscopes(v_i/c_i) is overloaded. If given a variable v_i as input, it returns the set of subscopes that contain v_i. If passed a constraint c_i, it returns the set of subscopes incident to c_i. The function cardinality(c_i/σ) is also overloaded. It takes as input either a constraint c_i or a subscope σ and returns the number of active variables in $scope(c_i)$ or σ. The function numActiveCons(v_i/σ) is overloaded. It takes as input either a variable v_i or subscope σ and returns the number of active constraints incident to v_i or σ, respectively. The function incidentCons(v_i) gives the set of constraints incident to a variable v_i. Finally, the function neighbors(c_i) gives the set of constraints that share at least one variable with c_i. Of course, at preprocessing, we initialize, to their appropriate values, the counters and sets accessed with the above-listed functions.

```
Algorithm 12: IsSubset($c_i$)                         Checks if $c_i$ can be removed
  Input: Constraint $c_i$ to check
  Output: Whether $c_i$ is a subset of another active constraint
1 if cardinality($c_i$) $\leq 1$ then
2     return true
3 for $\sigma \in$ incidentSubscopes($c_i$) do
4     if numActiveCons($\sigma$) $> 1$ and cardinality($c_i$) $\leq$ cardinality($\sigma$) then
5         return true
6 return false
```

Algorithm 12 takes a constraint c_i and uses Proposition 5 to determine if c_i is a subset of another active constraint. Line 1 is used to avoid looping through all the subscopes incident to the passed constraint (a constraint with a single active variable in its scope is guaranteed to be either a subset of another constraint or the only active constraint with that variable in its scope). The conditional numActiveCons(σ) > 1 in line 4 ensures that there is at least one other active constraint with the subscope σ.

```
Algorithm 13: RemoveConstraint($c_i$)                  Removes a constraint
  Input: Constraint $c_i$ to remove from problem
1 for $\sigma \in$ incidentSubscopes($c_i$) do
2     numActiveCons($\sigma$) $\leftarrow$ numActiveCons($\sigma$) $- 1$
3 for $v_i \in$ scope($c_i$) do
4     numActiveCons($v_i$) $\leftarrow$ numActiveCons($v_i$) $- 1$
5 activeCons $\leftarrow$ activeCons $\setminus \{c_i\}$
```

Algorithm 13 takes a constraint c_i that is known to be properly contained in another, updates the counts of any variables or subscopes incident to c_i, and finally removes c_i from activeCons. Similarly Algorithm 14 takes a variable v_i, updates the

```
Algorithm 14: RemoveVariable($v_i$)                     Removes a variable
  Input: Variable $v_i$ to remove from problem
  for $c_i \in$ incidentCons($v_i$) do
      if $c_i \in$ activeCons then
          cardinality($c_i$) $\leftarrow$ cardinality($c_i$) $- 1$
          queue $\leftarrow$ queue $\cup \{c_i\}$
  for $\sigma \in$ incidentSubscopes($v_i$) do
      cardinality($\sigma$) $\leftarrow$ cardinality($\sigma$) $- 1$
  activeVars $\leftarrow$ activeVars $\setminus \{v_i\}$
```

counts of active variables of the constraints and subscopes where v_i appears, and removes v_i from activeVars. Algorithm 14 also inserts any active constraints incident to v_i into the queue because those constraints may now be properly contained in other constraints following v_i's removal from the graph.

```
Algorithm 15: FindDangles($c_i$)          Finds all dangles in the subproblem
  Input: The variable $v_i$ that was just instantiated
  Output: The collections of dangles and removed variables
  dangles $\leftarrow \emptyset$, removedVars $\leftarrow \emptyset$, queue $\leftarrow \emptyset$
  removeVariable($v_i$)                               // Initializes queue
  while queue $\neq \emptyset$ do
      possibleLeaf $\leftarrow$ pop(queue)
      if possibleLeaf $\in$ activeCons and isSubset(possibleLeaf) then
          (dangle, $\Delta$) $\leftarrow$ grabDangle(possibleLeaf)
          dangles $\leftarrow$ dangles $\cup$ {dangle}
          removedVars $\leftarrow$ removedVars $\cup \Delta$
  return dangles, removedVars
```

Algorithms 15 and 16 provide an iterative version of the GYO Reduction and use Algorithms 12–14 to correctly maintain the various above-described counts. Algorithm 15 is executed after search instantiates a variable v_i. It returns the sets of dangles and variables removed following the instantiation of v_i. The call to Algorithm 14 on line 2 ensures that the counts of any subscopes and constraints incident to v_i are updated to reflect v_i's assignment and initializes the queue with any active constraints incident to v_i. Each constraint in the queue is now potentially properly contained in another due to either the removal or assignment of a variable. This check is done on line 5 of Algorithm 15 by calling Algorithm 12 after popping, in line 4, a constraint from the queue. If the popped constraint is contained in another, the constraint (now confirmed to be the leaf of a dangle) is passed, in line 6, as input to Algorithm 16 in order to collect all other constraints and variables that are attached to the leaf.

```
Algorithm 16: GrabDangle($c_i$)                                  Builds dangle leaf up
   Input: Constraint $c_i$ that is the leaf of a dangle
   Output: The dangle that begins with $c_i$ and any variables removed
1  dangle ← $\emptyset$, removedVars ← $\emptyset$, toVisit ← $\emptyset$
2  push($c_i$, toVisit)                                              // FIFO Queue
3  while toVisit $\neq \emptyset$ do
4      leaf ← pop(toVisit)
5      if isSubset(leaf) then
6          for neighbor $\in$ neighbors(leaf) do
7              if neighbor $\in$ activeCons then
8                  push((leaf, neighbor), dangle)
9                  push(neighbor, toVisit)
10         for $v_i \in$ scope(leaf) do
11             if $v_i$ is not instantiated and numActiveCons($v_i$) $\leq 2$ and
                 $v_i \in$ activeVars then
12                 removeVariable($v_i$)
13                 removedVars ← removedVars $\cup \{v_i\}$
14         removeConstraint(leaf)
15 return dangle, removedVars
```

Algorithm 16 takes as input a constraint that has already been recognized as a leaf. Lines 6–9 of Algorithm 16 add, to the dangle, the edges connecting the leaf to its parents/neighbors, where an edge is an ordered pair (leaf,neighbor). Importantly, this list is ordered to ensure that we can later enforce directional GAC or PWC along the dangles by simply linearly iterating along the returned dangle. Line 9 adds any neighbors of the leaf to a FIFO queue to check whether or not the leaf's parents can also be removed. Lines 10–13 check whether or not any variables in the scope of the constraint can also be removed (because they may now be constrained by only a single constraint). We check whether `numActiveCons`$(v_i) \leq 2$ because the number of active constraints for the variable will not be updated until we call `removeConstraint(leaf)` on line 14. Finally, in line 15, we return the dangle and any removed variables.

Proposition 6. *The complexity of Algorithm 15 is $O(|\mathcal{C}|^2 \cdot (|\mathcal{X}| + k))$, where k is the maximum arity of a constraint.*

Proof. Each time a constraint is popped from the queue, Algorithm 12 is called to check for hyperedge removal, which requires checking all of its incident subscopes. In the worst case of a fully connected graph where each subscope is unique, this operation requires $O(|\mathcal{C}|^2)$ operations. Each constraint could potentially enter the queue k times (once for each removed variable in its scope), resulting in a complexity of $O(|\mathcal{C}|^2 \cdot k)$. Each constraint can only be removed once, and removing a constraint requires at most $O(|\mathcal{C}|^2 + k)$ operations to decrement the appropriate counters, which is dominated by the complexity of calls to Algorithm 12. Removing a variable requires $O(|\mathcal{C}| + |\mathcal{C}|^2) = O(|\mathcal{C}|^2)$, to iterate over the incident constraints and subscopes, respectively. This happens once per variable. Thus the complexity of Algorithm 15 is $O((|\mathcal{X}| \cdot |\mathcal{C}|^2) + (|\mathcal{C}|^2 \cdot k) = O(|\mathcal{C}|^2 \cdot (k + |\mathcal{X}|))$. □

Despite the seemingly high complexity, our algorithm still works well in practice because the number of subscopes incident to either a constraint or variable rarely approaches the worst case calculated above. Additionally, we note that it is possible to specialize this algorithm to be used for binary CSPs because certain guarantees can be made regarding the counts of various structures used in the algorithm. However, we do not make use of any such specialization in our implementation and leave this refinement for future work.

On binary problems, our approach is equivalent to that presented by Sabin and Freuder [1997], barring differences induced by variable ordering heuristics. Both algorithms will remove variables with degrees of 1 from the problem, enforce a consistency over the remaining subproblem, and eventually add back the removed variables, solving them backtrack free. However, as previously mentioned, the work presented by Sabin and Freuder [1997] would not trivially extend to non-binary CSPs. Applying their technique to the dual graph would not ensure α-acyclicity and may miss removable portions of the problem as shown in Section 6.2.1.

6.4 Ensuring Satisfiability of Dangles

Like directional arc consistency ensures the consistency of a binary CSP [Dechter and Pearl, 1988; Freuder, 1982a], directional pairwise consistency ensures the consistency of a non-binary CSP [Beeri *et al.*, 1983]. We advocate to enforce directional pairwise consistency on the subproblems identified as dangles.

Algorithm 16 returns each dangle as an ordered list of dual edges from the leaves of the problem to the root. Thus, after Algorithm 15 completes, we need only perform a single pass of GAC (for binary constraints) or PWC (for non-binary constraints) on the constraints in each edge to ensure each removed dangle is satisfiable. If all dangles are found to be satisfiable, the search procedure removes those variables and constraints from the problem, and search continues as normal on the remaining cyclic subproblem.

We chose to use the PW-CT algorithm from Chapter 4.5 to enforce GAC and fPWC along the dangles, as it is currently the fastest known algorithm for enforcing fPWC, only enforces fPWC when necessary, and consumes only a marginal amount of additional memory compared to CompactTable. Enforcing directional consistency along the dangles using PW-CT required only minor modifications to the algorithm. Rather than enforcing PWC with respect to all subscopes incident to a constraint, the directional version of PW-CT instead only revises the dangles from the child nodes in the dangles to their parents without re-queueing constraints. The direction of enforcement is reversed when the cyclic portion the problem is solved to extend the solution to the dangles backtrack-free. It is crucial to enforce fPWC over the entire problem as a preprocessing step to ensure that any invalid tuples are removed from would-be dangles from the outset, lest dangle identification mistakenly assume one of these tuples is a valid support.

We note that our solver uses d-way branching, meaning that after an ordering heuristic chooses a variable, it attempts to find a solution with each value in the domain of the variable before backtracking. Thus, it is only necessary to *identify* dangles when variables are chosen by our heuristic and upon backtracking, not per assignment. However, we obviously must still enforce consistency along the dangles at each assignment because enforcing consistency may change whether or not the dangle contains a solution.

6.5 Empirical Evaluation of Dangle Identification

Experiments for dangle identification were run on 72 benchmarks consisting of non-binary CSPs from the CPAI08 dataset[2]. Binary benchmarks were excluded due to the theoretical equivalency of prior work in this area on binary constraints [Sabin and

[2]http://www.cril.univ-artois.fr/CPAI08/

Freuder, 1997]. A total of 2,023 instances were tested.

Five algorithms were chosen to test dangle identification: CompactTable, AllSolFB^{m3}, AllSolFB$^{\Psi}$,PerFB^{m3}-os-r, PerFB$^{\Psi}$-os-r. The m-wise consistency algorithms were chosen based on the results from Chapter 5 (i.e., the best performing algorithms from among those introduced in this dissertation were chosen). PW-CT is implicitly run alongside these algorithms when ensuring consistency of dangles, and was omitted from these experiments as enforcing PW-CT on top of dangle identification would be somewhat redundant; the only savings would be from enforcing directional PW-CT along dangles and removing the constraints in dangles from consideration.

The CompactTable, PerFB and AllSolFB algorithms were slightly modified to accommodate dangle identification. Previous to this work, Stampede had no mechanism for removing constraints from the problem after the initial construction of objects. Each of the algorithms was modified to incorporate the state of a constraint (i.e., whether the constraint was hidden or not) into its queueing mechanisms. In this work, this simply meant preventing hidden constraints from being enqueued.

The versions of these algorithms that make use of dangle identification are denoted with the prefix “Di-”. Additionally, we show the performance of a naive approach to dangle identification which only removes constraints that have a single uninstantiated variable in their scope. This approach does not require enforcement of PWC *so long as the removed constraints are made GAC prior to removal.* These variants serve as a baseline to show the relative improvement that enforcing PWC along dangles has compared to a naive approach, and are denoted with the prefix “H-”. The purpose of the naive approach is to identify the impact that identifying the dangles has, so the preprocessing steps for initializing dangle identification is kept, including an initial round of PWC enforcement. Any tuple or value removals from this initial round of

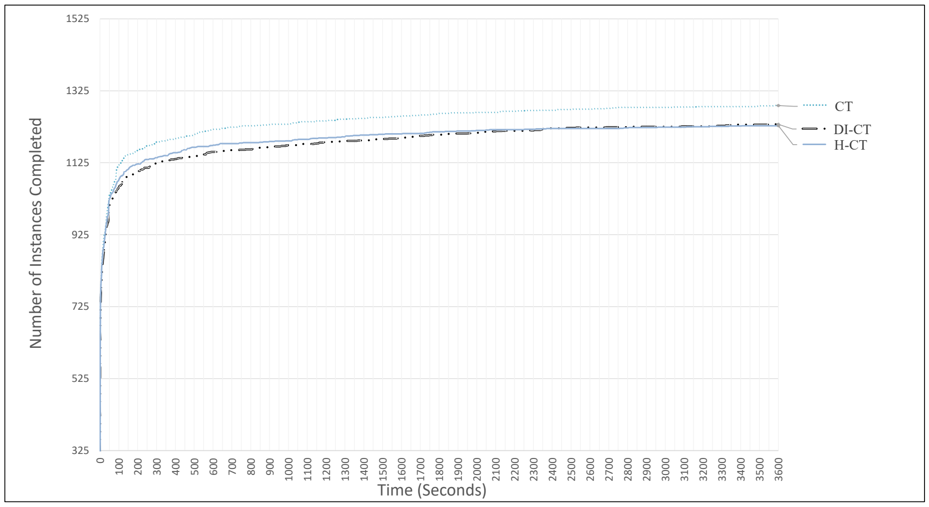
PWC enforcement are kept localized to dangle identification and do not impact other propagators.

Search over each CSP was performed using STAMPEDE to find the first consistent solution, using the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. The solver was limited to one hour and 8GB of memory. When an algorithm timed out or ran out of memory (OOM), it is recorded as having taken the full hour.

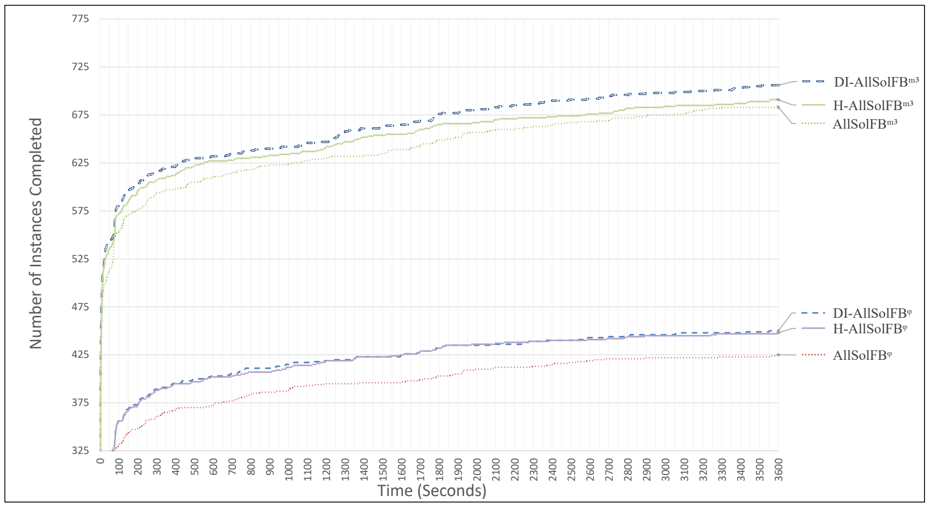
The charts in Figure 6.4 show the number of instances completed within a certain amount of time for all tested algorithms. The results for COMPACTTABLE in Figure 6.4a are somewhat disappointing. Using dangle identifications results in fewer instances being solved, even for the naive approach. This would imply that the decrease in performance is not due to the increased overhead from PW-CT. One potential source for the decrease in performance is the required enforcement of fPWC as a preprocessing step for dangle identification incurred by both versions. Our implementation uses the minimal dual graph for this preprocessing step[3].

Indeed both the naive and full versions of dangle identification run out of memory on 40 instances that COMPACTTABLE finishes. However, this does not fully explain the difference in performance, as H-CT also times out on 23 instances that COMPACTTABLE finishes, and Di-CT times out on 18 instances that COMPACTTABLE finishes. Restricting the instances to only those 23 instances that H-CT failed to complete reveals no discernible pattern; they're a mixture of random and structured instances. In fact, of those 23 instances, Di-CT finishes 11 of them (and does so faster than COMPACTTABLE on three problems). The only explanation for this behavior is the variable ordering heuristic causing H-CT to explore sub-optimal portions of the

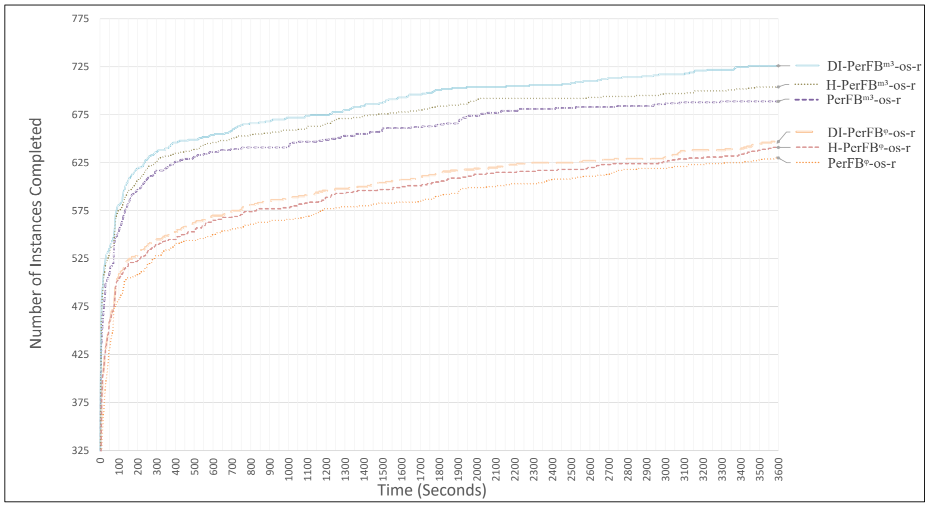
[3]This was an oversight when setting up the experiments; traditionally PWC algorithms benefit significantly from the minimal dual graph, but PW-CT sees relatively minor gains when its enforced over the minimal dual graph, and could have been omitted to prevent running out of memory on problems like BddLarge.



(a) CompactTable



(b) AllSolFB



(c) PerFB

Figure 6.4: Cumulative charts for dangle identification using $\frac{|\texttt{dom}|}{\texttt{wdeg}}$.

search tree, since it is guaranteed to do no more work than Di-CT, but may remove only a subset of the constraints that Di-CT removes, which could cause the search trees to diverge (and evidently somewhat considerably).

The results for PerFB and AllSolFB are more promising. Di-AllSolFB^{m3} consistently solves more problems than either H-AllSolFB^{m3} or AllSolFB^{m3}, and the results are even more pronounced for Di-PerFB^{m3}-os-r. The gains for AllSolFB$^{\Psi}$ and PerFB$^{\Psi}$ are present, but less pronounced, especially relative to H-AllSolFB$^{\Psi}$ and H-PerFB^{m3}-os-r. This is not terribly surprising in hindsight; enforcing minimality over an entire cluster forces a lot of work at each node of the search tree, and causes far fewer nodes to be explored. One of the key benefits of dangle identification is reducing the number of nodes visited, the impact of which is dwarfed by enforcing cluster minimality. Another key benefit of dangle identification is reducing the amount of work required at every node expanded, but again, this impact is dampened if the propagator causes fewer nodes to be explored.

The t-tests in Table 6.1 confirm these findings. As in previous chapters, cells in the table are labeled "True" if there is a statistically significant difference between the algorithms in the row and column, and the algorithm corresponding to the row had a lower mean solving time than the algorithm in the column. The dashed lines in the table show the boundaries of a group of algorithms (e.g., all of the AllSolFB algorithms are in the top left of the table). With the exception of CompactTable, the dangle identification versions of the tested algorithms outperform their standard counterparts. There is no statistical difference between the naive approach and full dangle identification when enforcing cluster minimality, but full dangle identification is shown to be an improvement for combinations of size three.

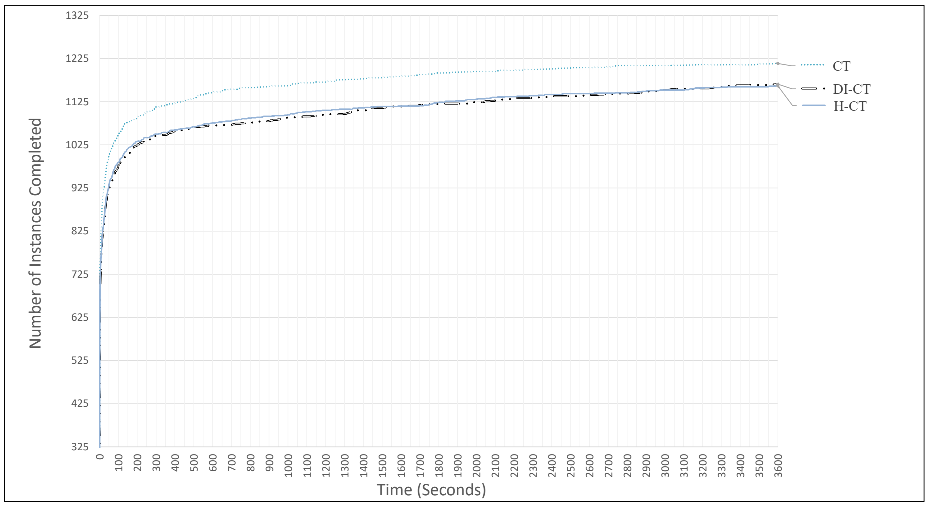
Table 6.2 shows the number of instances that algorithms in each row were able to complete that the corresponding algorithms in the table's columns were not able to

Table 6.1: Pairwise t-test results for Dangle Identification $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. Cells are labeled "True" if the results were significantly different (p <.05) and the value of the row entry was less than the column.

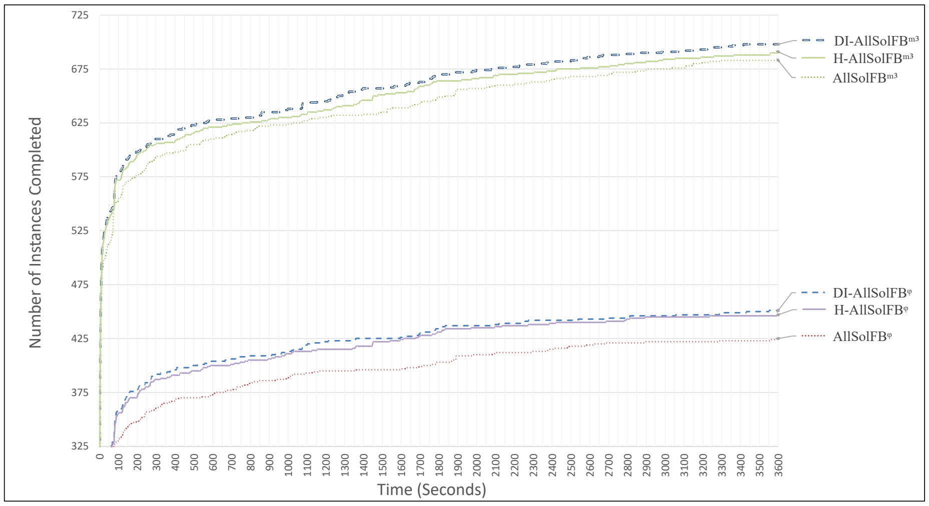
	AllSolFB$^{\Psi}$	H-AllSolFB$^{\Psi}$	Di-AllSolFB$^{\Psi}$	AllSolFB^{m3}	H-AllSolFB^{m3}	Di-AllSolFB^{m3}	CT	H-CT	Di-CT	PerFB$^{\Psi}$	H-PerFB$^{\Psi}$-os-r	Di-PerFB$^{\Psi}$-os-r	PerFB^{m3}	H-PerFB^{m3}-os-r	Di-PerFB^{m3}-os-r
AllSolFB$^{\Psi}$	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
H-AllSolFB$^{\Psi}$	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Di-AllSolFB$^{\Psi}$	T	-	-	-	-	-	-	-	-	-	-	-	-	-	-
AllSolFB^{m3}	T	T	T	-	-	-	-	-	-	T	T	T	-	-	-
H-AllSolFB^{m3}	T	T	T	-	-	-	-	-	-	T	T	T	-	-	-
Di-AllSolFB^{m3}	T	T	T	T	T	-	-	-	-	T	T	T	-	-	-
CT	T	T	T	T	T	T	-	T	T	T	T	T	T	T	T
H-CT	T	T	T	T	T	T	-	-	-	T	T	T	T	T	T
Di-CT	T	T	T	T	T	T	-	-	-	T	T	T	T	T	T
PerFB$^{\Psi}$-os-r	T	T	T	-	-	-	-	-	-	-	-	-	-	-	-
H-PerFB$^{\Psi}$-os-r	T	T	T	-	-	-	-	-	-	-	-	-	-	-	-
Di-PerFB$^{\Psi}$-os-r	T	T	T	-	-	-	-	-	-	T	-	-	-	-	-
PerFB^{m3}-os-r	T	T	T	T	-	-	-	-	-	T	T	T	-	-	-
H-PerFB^{m3}-os-r	T	T	T	T	T	-	-	-	-	T	T	T	-	-	-
Di-PerFB^{m3}-os-r	T	T	T	T	T	T	-	-	-	T	T	T	T	T	-

complete. Again, the dashed lines show the boundaries for groups of the underlying algorithm. Notably, *every* version of an algorithm is able to complete some instances that neither of the other two variants were able to complete (e.g., H-CT and Di-CT were able to complete 5 and 9 instances, respectively, that CompactTable timed out on).

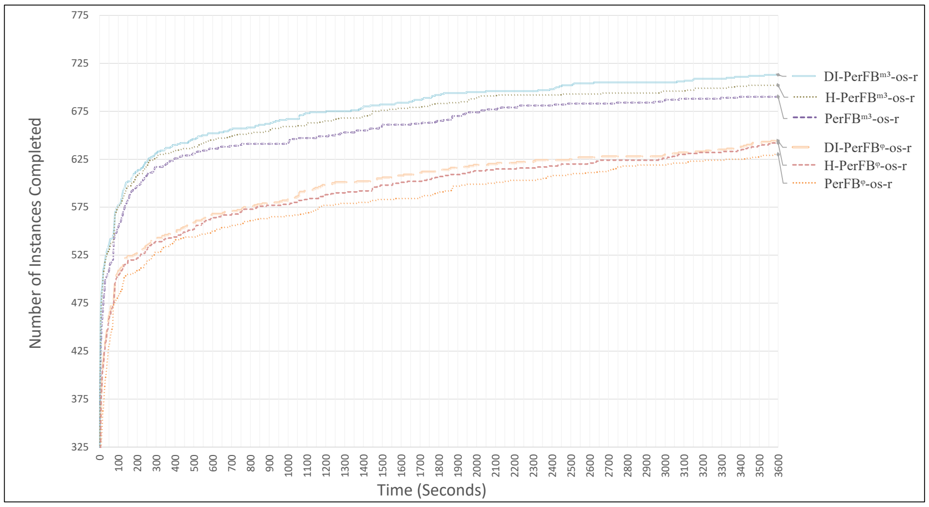
While we believe the results presented above clearly show the advantage of DI for certain algorithms, the variability in CompactTable is almost certainly due to the unpredictability introduced by the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic. These experiments were also run using $\frac{|\texttt{dom}|}{\texttt{ddeg}}$, which has previously been shown to be a more stable heuristic



(a) CompactTable



(b) AllSolFB



(c) PerFB

Figure 6.5: Cumulative charts for dangle identification using $\frac{|\texttt{dom}|}{\texttt{ddeg}}$.

Table 6.2: $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ results showing the number of instances algorithms in each row were able to complete that algorithms in each column were unable to finish due to hitting the memory limit (OOM) or a timeout (TO).

	AllSolFB$^{\Psi}$		H-AllSolFB$^{\Psi}$		DI-AllSolFB$^{\Psi}$		AllSolFB^{m3}		H-AllSolFB^{m3}		DI-AllSolFB^{m3}		CT		H-CT		DI-CT		PerFB$^{\Psi}$-os-r		H-PerFB$^{\Psi}$-os-r		DI-PerFB$^{\Psi}$-os-r		PerFB^{m3}-os-r		H-PerFB^{m3}-os-r		DI-PerFB^{m3}-os-r	
	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO	OOM	TO
AllSolFB$^{\Psi}$	-	-	0	16	0	18	1	7	1	22	1	22	0	9	0	10	0	10	0	1	0	16	0	17	1	8	1	22	1	20
H-AllSolFB$^{\Psi}$	0	31	-	-	0	5	1	25	1	10	1	13	0	1	0	2	0	0	0	22	0	1	0	7	1	26	1	11	1	12
DI-AllSolFB$^{\Psi}$	0	34	0	5	-	-	1	27	1	14	1	8	0	1	0	1	0	0	0	25	0	6	0	2	1	28	1	15	1	7
AllSolFB^{m3}	0	259	0	262	0	264	-	-	0	25	0	24	0	11	0	12	0	12	0	99	0	106	0	107	10	1	10	23	10	20
H-AllSolFB^{m3}	0	281	0	253	0	259	0	32	-	-	0	5	0	1	0	2	0	0	0	125	0	96	0	101	10	34	9	1	10	6
DI-AllSolFB^{m3}	0	294	0	269	0	265	0	44	0	18	-	-	0	1	0	1	0	0	0	137	0	111	0	102	11	47	11	19	11	0
CT	84	767	86	742	86	741	86	491	89	465	89	450	-	-	40	23	40	18	86	563	88	538	87	536	189	410	192	384	191	361
H-CT	46	750	47	726	47	724	49	473	50	449	50	433	0	5	-	-	0	9	49	545	49	522	48	519	152	392	153	367	152	344
DI-CT	46	755	47	728	47	727	50	477	51	450	51	435	0	9	0	15	-	-	50	549	50	523	49	521	150	399	151	372	151	348
PerFB$^{\Psi}$-os-r	0	192	0	198	0	201	0	37	0	61	0	58	0	10	0	11	0	11	-	-	0	29	0	29	4	34	5	53	5	48
H-PerFB$^{\Psi}$-os-r	0	216	0	187	0	193	0	61	0	40	0	41	0	1	0	2	0	0	0	41	-	-	0	8	4	56	5	33	5	31
DI-PerFB$^{\Psi}$-os-r	0	223	0	198	0	194	0	65	0	46	0	34	0	1	0	1	0	0	0	44	0	13	-	-	4	60	5	40	5	24
PerFB^{m3}-os-r	0	258	0	262	0	264	0	5	0	32	0	29	0	10	0	11	0	11	0	92	0	103	0	103	-	-	0	27	0	24
H-PerFB^{m3}-os-r	0	286	0	261	0	266	0	42	0	12	0	14	0	1	0	1	0	0	0	123	0	94	0	97	1	40	-	-	1	5
DI-PerFB^{m3}-os-r	0	306	0	283	0	279	0	63	0	35	0	13	0	1	0	1	0	0	0	141	0	112	0	102	0	62	0	27	-	-

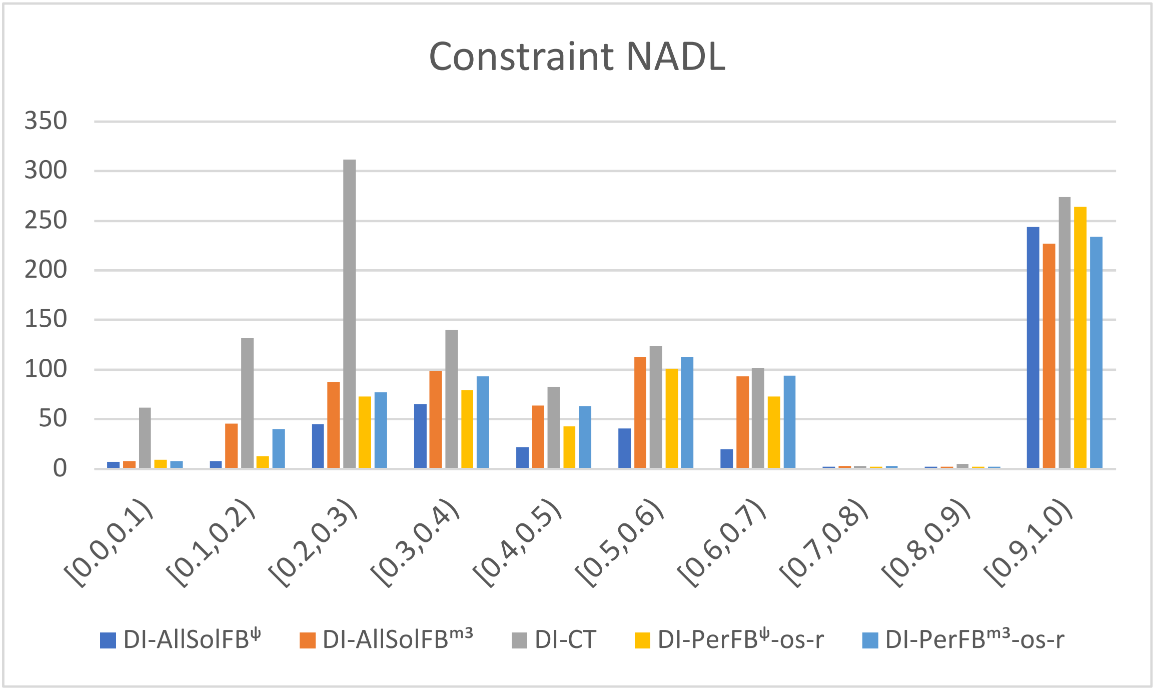
when used with relational consistencies [Balafrej *et al.*, 2015; Paparrizou and Stergiou, 2016, 2017]. Figure 6.5 shows the cumulative run times for the same set of algorithms using $\frac{|\texttt{dom}|}{\texttt{ddeg}}$. While the number of instances solved for each algorithm is reduced when using $\frac{|\texttt{dom}|}{\texttt{ddeg}}$, the relative performance of the algorithms remain largely unchanged.

Looking more closely, if we compare $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ and $\frac{|\texttt{dom}|}{\texttt{wdeg}}$, looking at the number of nodes visited on instances CompactTable and Di-CT both completed, we can see the variance with $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ is actually *higher* than with $\frac{|\texttt{dom}|}{\texttt{wdeg}}$. Roughly 14% of instances using $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ visited more nodes using Di-CT than CompactTable, with a mean percentage difference of 83% (and a mean of 1,912,885 additional nodes visited) on those instances where Di-CT visited more nodes. The $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ heuristic, on the other hand, only visited more nodes with Di-CT in 9.5% of instances where both algorithms finished, with a mean difference of 47% (and a mean of 948,669 additional nodes visited). We believe this provides clear motivation for further investigation into ordering heuristics that are robust to relational consistencies and the identification and removal of tractable subproblems.

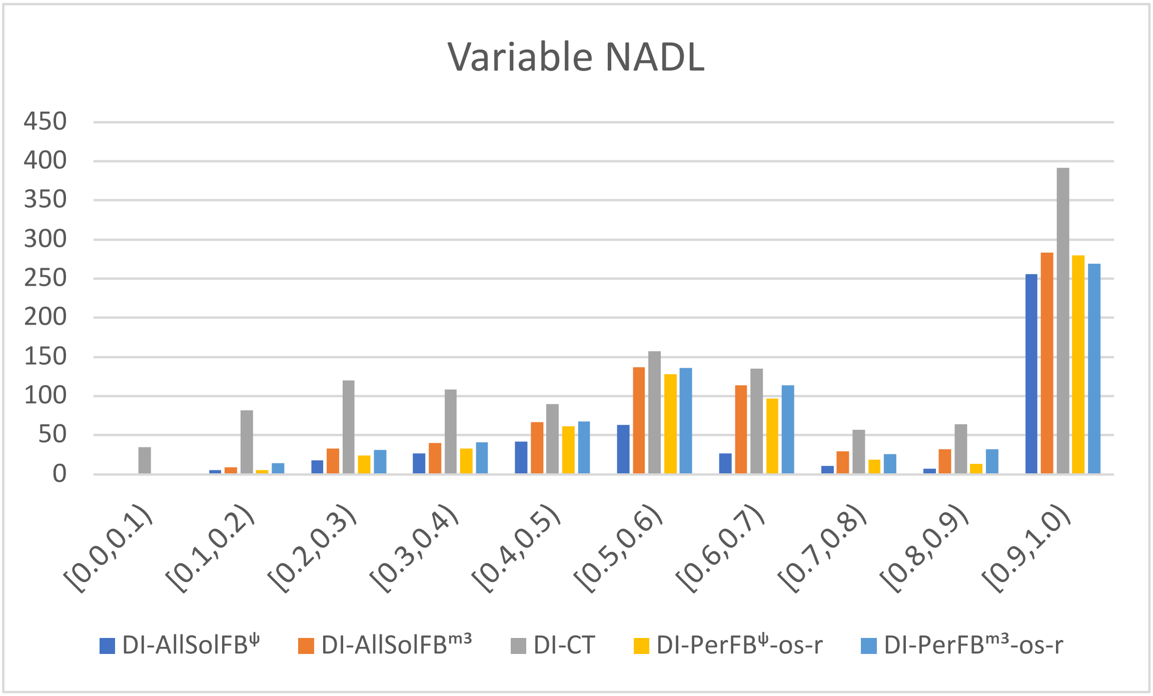
Geschwender [2018] provided two metrics to help analyze the performance of dangle identification on the dual graph during the subproblem search in PerTuple and AllSol. The first of these statistics is the Normalized Average Dangle-Level (NADL), which measures the depth that dangles are identified during search. The value is calculated by tracking the number of dangling vertices at each level of search and averaging the values when search ends. The value is then normalized by dividing by the depth of the search tree (i.e., the number of variables in the graph in the case of the dangle identification procedure described in this chapter). The value ranges from 0, indicating all dangles identified during search were identified at preprocessing, to $\frac{n-2}{n}$, where n is the depth of the search tree. The second value is the Average Percent Dangles Identified (ADPI), which measures the average number of future variables or constraints that are identified as part of a dangle at each visited node in search.

Figures 6.6 and 6.7 provide histograms of the NADL and ADPI for both constraints and variables. The values for each algorithm are displayed. The large variance between algorithms is due to the incorporation of all completed instances for each algorithm in to the count in each range in the histogram; in fact, the NADL and ADPI are identical for AllSol and PerTuple algorithms when operating on the same sizes of combinations.

The NADL for both variables and constraints show that in many cases, all dangles are identified at the end of search, though a fair amount (especially in the case of dangling constraints in CompactTable) are identified quite early in search. The ADPI for constraints typically falls between 0% and 5%, and between 0% and 3%

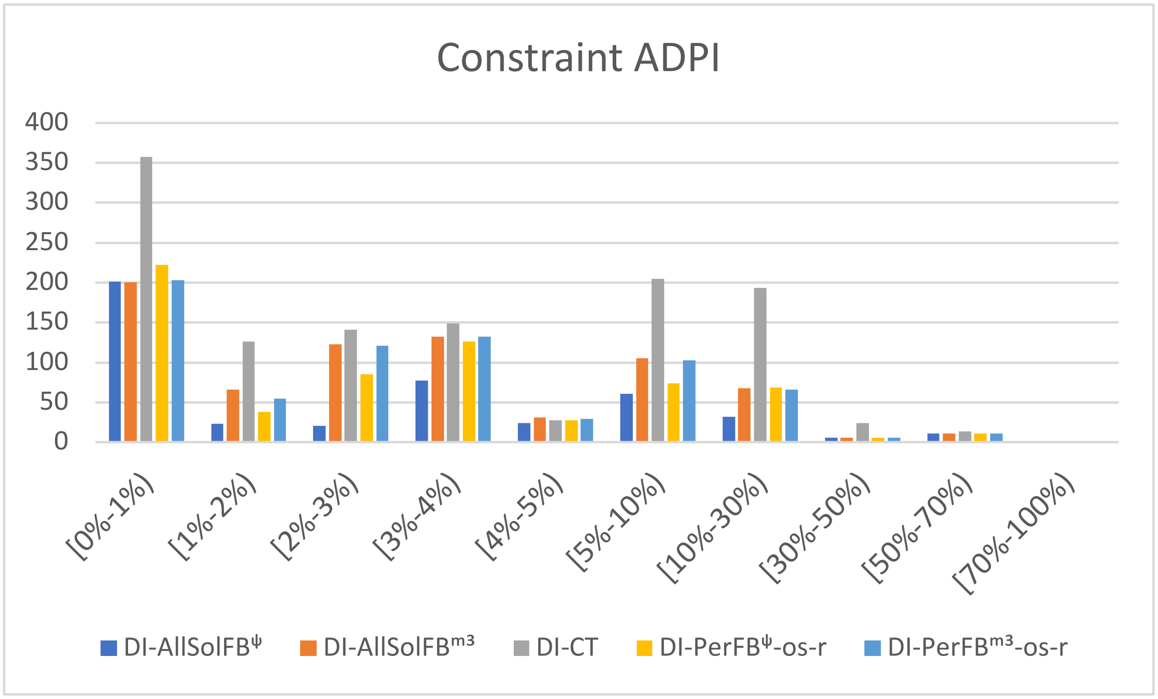


(a) Constraint NADL

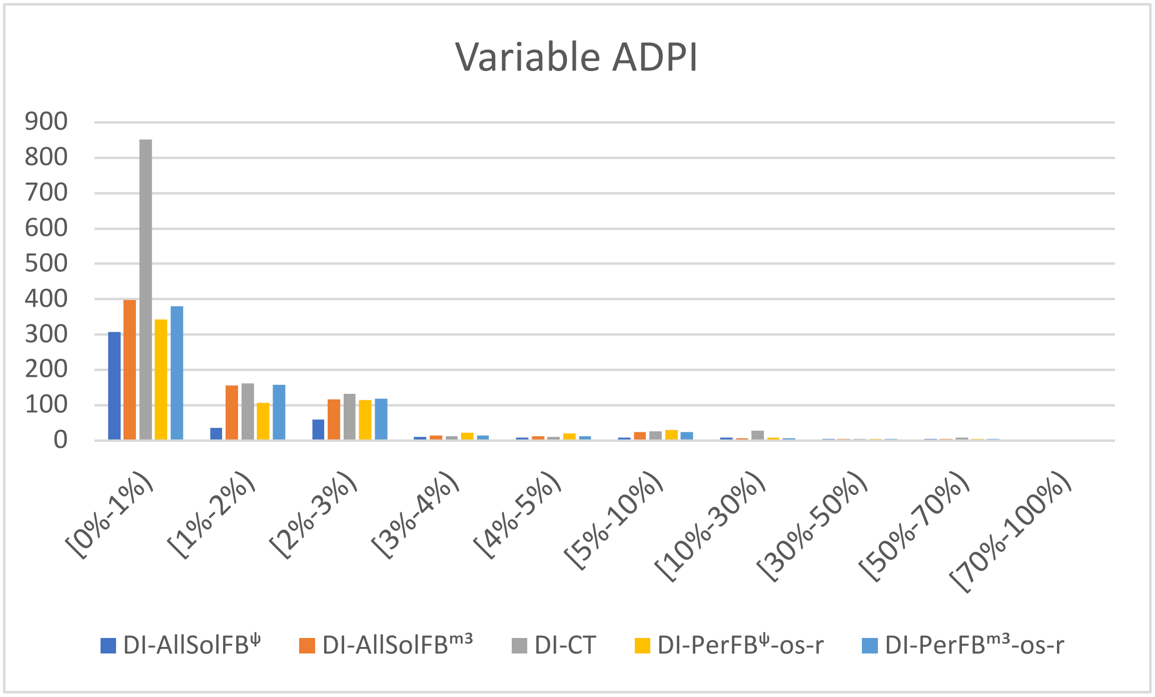


(b) Variable NADL

Figure 6.6: Histograms for the NADL metrics for constraints and variables.



(a) Constraint ADPI



(b) Variable ADPI

Figure 6.7: Histograms for the ADPI metrics for constraints and variables.

for variables (note the uneven scaling in the buckets, which was used to give more visibility to the breakdown at lower percentages). While these are relatively low values, it shows the potential strength of the dangle identification technique: Although a low percentage of variables and constraints were removed from the problem in most instances, and at a fairly deep level of search, we still see significant improvement in the relational consistency algorithms when using dangle identification. This emphasizes the need for specialized ordering heuristics that could be used to facilitate the creation of dangles dynamically during search, rather than relying on general purpose heuristics like the $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ ordering heuristic.

Summary

In this chapter, we established that using a minimal dual graph during search to identify dangles by iteratively identifying vertices of degree 1 can lead to incorrect results, and provided justification for the choice of using the GYO reduction for determining acyclicity on non-binary problems. Further, we designed an efficient algorithm for identifying dangles that applies the GYO reduction during search (i.e., as variables are instantiated), which is simple enough that it can be easily integrated into existing search procedures. We demonstrated the benefits of dangle identification for certain propagators, and provided further evidence for the need to for more robust and HLC-friendly ordering heuristics.

Beyond investigation into ordering heuristics, the work presented in this chapter can be extended in a few ways. Enforcing PW-CT along dangles is a task well suited for parallelization. Search could conceivably begin enforcing consistency on the subproblem after identifying the structural dangles concurrently with the enforcement of PW-CT on the dangles. If either PW-CT or the subproblem finds an inconsistent state, the other thread could be pre-empted, mitigating the additional cost imposed by enforcing PW-CT. Enforcing PW-CT on some problems is still prohibitive due to the large number of tuples and high degree of the problems, and this approach would be especially advantageous in those circumstances.

Further improvements could also be made to the propagators themselves to better incorporate the hidden constraint state into their operation. AllSolFB and PerFB, for example, could skip hidden constraints in their tuple search, though the theoretical level of consistency enforced by such a modification would need to be evaluated in depth.

Chapter 7

Conclusions

We conclude the dissertation by summarizing our contributions and discussing avenues for future research.

7.1 Summary of Contributions

We began this dissertation seeking to establish the promise and practicality of high level consistencies in CSPs. We accomplished this goal by designing and implementing a solver, STAMPEDE, capable of easily incorporating new propagators into its operation, including relational consistencies. The creation of STAMPEDE has aided in the development of several novel approaches to integration relational consistencies with search [Geschwender *et al.*, 2016; Geschwender, 2018; Woodward *et al.*, 2018; Woodward, 2018], examination of performance on different models of nonogram puzzles using HLC and GAC [Tran, 2019], visualizations to better understand the impact of propagators on search [Howell *et al.*, 2020; Woodward *et al.*, 2018; Woodward, 2018], and new consistency algorithms responsible for pushing the state-of-the-art [Schneider and Choueiry, 2018; Woodward *et al.*, 2017]. The specific manner in which STAMPEDE helped to unlock this research was outlined in Chapter 3.

Table 7.1 summarizes the novel algorithms introduced by this dissertation.

In Chapter 4, we outlined ways of making PWC algorithms more effective, and applied those methods to three previously existing algorithms, eSTR2, eSTR2^w, and PW-AC, greatly enhancing their performance. We then created a new algorithm,

Table 7.1: Algorithms introduced in this thesis.

Algorithm	Consistency Property
AllSolFB	m-wise
PW-AC2	PWC
PW-CT	fPWC
Directional PW-CT	Partial fPWC
PerFB	m-wise
Dangle Identification	Search procedure

PW-CT, that incorporated the advances in PWC algorithms with the state-of-the-art for enforcing GAC, and showed that PW-CT is far and away the best algorithm for enforcing PWC.

In Chapter 5, we extended the lessons learned from PWC to algorithms for enforcing R($*$,m)C, and again showed that the resulting algorithms, PerFB and AllSolFB, are superior to other algorithms used for enforcing R($*$,m)C.

Finally, in Chapter 6, we used PW-CT to enforce minimality on dynamically discovered tractable subproblems of CSPs during search and showed it to be beneficial for relational consistencies in particular.

7.2 Future Work

Below we identify potential directions for future work.

1. *Better ordering heuristics for HLC and relational consistencies*: As shown in Chapter 6, the most commonly used ordering heuristics, $\frac{|\texttt{dom}|}{\texttt{ddeg}}$ and $\frac{|\texttt{dom}|}{\texttt{wdeg}}$, are unpredictable when applied to non-traditional techniques. Both heuristics can explore radically different trees when dangle identification removes constraints from the problem. This is also a problem for HLC and relational consistencies with respect to $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ since the constraint weight to increment upon detecting an inconsistency when enforcing relational consistencies is not obvious. Inves-

tigating potential alternatives to these ordering heuristics that are more robust and stable while maintaining or improving upon the performance of $\frac{|\texttt{dom}|}{\texttt{wdeg}}$ would be immensely helpful for research, as the current heuristics can be too erratic to make drawing conclusions about the performance of two or more algorithms fraught.

2. *Integrating* PW-CT *into R(∗,m)C*: We were able to show large improvements when using bitsets and lazy enforcement of PWC in our algorithm PW-CT. It is not inconceivable that techniques similar to those used for PW-CT could be incorporated into PerFB or AllSolFB to achieve similar gains in performance.

3. *Investigate the use of GPUs to power relational consistencies*: Several of the algorithms presented in this dissertation are potentially good candidates for GPUs. PW-CT, in particular, could likely be transformed into a set of matrix multiplications using the bitsets introduced by CompactTable. If that avenue proves fruitful, extending it to R($*$,m)C would likely show significant benefits as the core operation of PerTuple is embarrassingly parallelizable (i.e., all of the tuple searches in a given combination can be executed independently).

4. *Better incorporate dangle identification into consistency algorithms and* Stampede: The current implementation of dangle identification in Stampede marks a boolean in the constraints and variables indicating whether they have been removed from the problem. This was an ad-hoc solution to incorporate removals of constraints due to dangle identification into a select group of propagators. Stampede should be extended to support this mechanism for *arbitrary* propagators, at least to some extent. Some propagators, such as those used to

enforce R($*$,m)C, could further benefit from dangle identification by altering which constraints they incorporate in their subproblem searches.

5. *Integrate all recent innovations for improving relational consistencies*: Alongside this dissertation, the Constraint Systems Laboratory has published a corpus of work describing a multitude of ways that relational consistencies can be made more viable. Isolating enforcement of consistencies along specific structures, partially enforcing them, and improvements to the search performed over combinations in PerTuple and AllSol are just a few of the directions that have been explored and empirically verified. Unifying these approaches into new hybrid algorithms could be a viable alternative to the current state-of-the-art GAC propagators.

6. *Extended relational propagators in* Stampede *to support intension constraints*: Most, if not all, of the algorithms currently implemented in Stampede implicitly assume they're operating on table constraints. However, it seems to be a somewhat straightforward task to offer an abstraction layer on top of other types of constraints that make them compatible with algorithms that operate on table constraints without fully enumerating the allowed tuples (i.e., the combinations of variable-value assignments the constraint supports). For example, a sparse bit set could be used to store the indices of the valid combinations, rather than explicitly enumerating all tuples. This approach could be extended to table constraints as well to limit the memory overhead of storing table constraints, though approaches such as MDDs could also be explored in both cases.

7. *Use concurrency in dangle identification to amortize the cost of enforcing PWC on dangles*: One of the largest hits to performance in dangle identification is enforcing PWC along dangles, even when using PW-CT. If running GAC

on the the cyclic portion of the subproblem would expose an inconsistency, the extra effort used to enforce PW-CT along dangles is wasted. This could be mitigated by parallelizing the enforcement of PW-CT along dangles while another propagator enforces its consistency on the remaining cyclic portion of the CSP, terminating early if either thread discovers an inconsistency.

The work presented in this dissertation has succeeded in showing that relational consistencies can be a pragmatic choice for CSP solving by creating a solver tailor made for the incorporation of HLC and relational consistencies and by pushing the state-of-the-art forward for multiple relational consistency properties.

Appendix A

Per-Benchmark Results for GAC Algorithms

Tables A.1 and A.2 show detailed results for GAC algorithms used in this dissertation on binary and non-binary problems, respectively.

Table A.1: Aggregated instances results for GAC algorithms for all tested binary benchmarks.

		# Solved				ΣCPU (sec)			
	# Instances	CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
BH-4-13	7	0	0	0	0	0.0	0.0	0.0	0.0
BH-4-4	10	10	10	10	10	203.2	411.9	670.8	206.2
BH-4-7	20	0	0	0	0	0.0	0.0	0.0	0.0
QCP-10	15	15	15	15	15	3.4	3.4	4.4	3.7
QCP-15	15	15	15	15	15	520.3	591.2	1,049.3	607.6
QCP-20	15	5	5	4	5	1,153.2	1,312.8	>3,830.0	1,404.9
QCP-25	15	1	1	1	1	14.1	14.0	16.3	15.2
QWH-10	10	10	10	10	10	1.7	1.6	1.9	1.9
QWH-15	10	10	10	10	10	18.5	18.7	26.9	20.6
QWH-20	10	9	9	9	9	1,020.8	1,119.2	2,003.6	1,202.8
QWH-25	10	0	0	0	0	0.0	0.0	0.0	0.0
bqwh-15-106	100	100	100	100	100	21.1	22.5	34.4	25.1
bqwh-18-141	100	100	100	100	100	335.9	406.4	704.3	388.5
coloring	22	22	22	22	22	378.8	403.5	671.4	415.2
composed-25-1-2	10	10	10	10	10	0.8	1.0	1.7	1.2
composed-25-1-25	10	10	10	10	10	1.0	1.2	2.1	1.3

Table A.1 (continued)

		# Solved				ΣCPU (sec)			
	# Instances	CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
composed-25-1-40	10	10	10	10	10	1.0	1.3	2.4	1.4
composed-25-1-80	10	10	10	10	10	1.2	1.5	2.8	1.6
composed-25-10-20	10	10	10	10	10	2.1	2.3	3.2	2.7
composed-75-1-2	10	10	10	10	10	2.4	2.8	4.7	3.4
composed-75-1-25	10	10	10	10	10	2.6	3.2	5.7	3.7
composed-75-1-40	10	10	10	10	10	2.7	3.4	6.2	3.8
composed-75-1-80	10	10	10	10	10	2.8	3.4	6.0	3.7
domino	24	22	22	21	21	698.3	1,080.1	>4,567.0	>4,567.2
driver	7	7	7	7	7	64.4	61.5	88.4	73.3
ehi-85	100	100	100	100	100	251.3	257.5	354.7	294.1
ehi-90	100	100	100	100	100	270.7	271.2	365.8	311.8
fapp-fapp01	11	0	0	0	0	0.0	0.0	0.0	0.0
frb30-15	10	10	10	10	10	8.6	15.1	34.2	12.6
frb35-17	10	10	10	10	10	86.7	162.6	390.0	130.6
frb40-19	10	10	10	10	10	687.3	1,294.0	3,128.2	1,024.4
frb45-21	10	10	8	6	10	6,051.8	>13,487.0	>24,504.0	8,860.8
frb50-23	10	6	2	2	2	13,727.5	>15,050.0	>15,970.6	>14,901.7
frb53-24	10	2	0	0	2	3,035.3	>7,200.0	>7,200.0	4,464.3
frb56-25	10	0	0	0	0	0.0	0.0	0.0	0.0
frb59-26	10	0	0	0	0	0.0	0.0	0.0	0.0
geom	100	100	100	100	100	1,197.4	2,443.4	6,750.3	1,766.2
graphColoring-hos	14	9	12	10	9	>10,887.9	1,133.4	>7,609.8	>10,892.4
graphColoring-insertion-full-insertion	41	34	33	32	32	6,043.8	>7,558.4	>10,739.2	>8,871.2
graphColoring-insertion-k-insertion	32	17	16	16	17	1,662.9	>3,807.2	>3,927.2	2,132.0
graphColoring-leighton-leighton-15	28	9	8	7	8	3,167.4	>6,379.7	>10,167.8	>5,466.3

Table A.1 (continued)

	# Instances	# Solved				ΣCPU (sec)			
		CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
graphColoring-leighton-leighton-25	32	6	6	6	6	100.7	103.7	113.1	111.5
graphColoring-leighton-leighton-5	8	8	8	8	8	52.3	47.7	55.4	54.0
graphColoring-mug	8	4	4	4	4	0.1	0.1	0.1	0.1
graphColoring-myciel	16	13	13	13	13	1,460.2	1,787.9	2,828.7	1,639.6
graphColoring-register-fpsol	37	5	5	5	5	72.2	85.5	87.0	89.8
graphColoring-register-inithx	32	5	5	5	5	155.5	174.7	180.5	186.2
graphColoring-register-mulsol	49	9	9	9	9	56.3	79.6	77.1	84.0
graphColoring-register-zeroin	31	6	6	6	6	40.7	60.2	56.7	63.3
graphColoring-school	8	3	3	3	3	85.3	87.2	90.5	92.5
graphColoring-sgb-book	26	24	23	22	23	5,440.2	>8,934.9	>13,619.0	>7,592.2
graphColoring-sgb-games	4	4	4	4	4	171.0	242.4	562.4	216.1
graphColoring-sgb-miles	42	11	10	10	11	1,848.1	>3,743.3	>3,729.2	2,405.9
graphColoring-sgb-queen	50	16	15	15	15	4,812.0	>6,161.4	>10,101.3	>5,357.3
hanoi	5	5	5	5	4	2.7	35.7	2.5	>3,602.4
haystacks	51	2	2	2	2	1.0	1.1	1.9	1.3
jobShop-e0ddr1	10	7	5	5	7	3,686.9	>7,526.2	>9,257.3	4,572.2
jobShop-e0ddr2	10	6	5	4	6	1,476.0	>4,119.8	>7,248.9	2,664.1
jobShop-enddr1	10	9	9	9	9	88.0	169.0	169.5	184.1
jobShop-enddr2	6	4	4	3	4	892.8	1,402.0	>3,640.6	1,668.1
jobShop-ewddr2	10	10	10	10	10	127.4	272.6	142.7	301.9
knights	19	10	10	10	10	118.7	460.6	2,262.7	576.7
langford	4	4	4	4	4	27.7	43.9	93.4	36.8
langford2	24	16	16	16	16	61.7	108.5	95.6	88.1
langford3	23	16	15	14	15	2,682.5	>4,984.9	>9,873.7	>4,758.6
langford4	24	14	13	12	13	1,794.6	>4,306.7	>8,960.3	>4,186.3

Table A.1 (continued)

	# Instances	# Solved				ΣCPU (sec)			
		CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
lard	10	10	10	10	10	444.5	717.9	475.0	565.9
marc	10	10	10	10	10	456.7	783.3	488.8	692.8
os-taillard-4	30	30	30	30	30	246.3	643.9	2,194.8	482.8
os-taillard-5	30	19	15	13	18	8,818.6	>17,033.9	>28,115.6	>11,082.0
os-taillard-7	30	1	7	6	7	>22,214.3	5,459.1	>11,459.4	5,171.0
pigeons	25	13	13	13	13	1,869.3	1,786.2	3,110.6	2,056.1
queenAttacking	10	4	4	4	4	156.3	288.9	668.7	201.4
queens	14	9	9	9	9	51.2	95.4	73.4	106.9
queensKnights	18	11	11	10	11	467.2	1,107.2	>5,490.3	1,091.8
rand-2-23	10	10	10	10	10	1,071.2	1,925.2	4,083.5	1,621.8
rand-2-24	10	10	10	10	10	2,383.3	4,295.6	9,005.2	3,587.7
rand-2-25	10	10	10	10	10	4,840.6	8,690.1	18,018.4	7,244.0
rand-2-26	10	10	3	3	10	14,471.8	>28,766.1	>32,482.5	21,478.0
rand-2-27	10	4	1	1	4	5,666.1	>12,298.9	>13,822.4	8,380.3
rand-2-30-15-fcd	50	50	50	50	50	55.5	96.9	217.9	80.6
rand-2-30-15	50	50	50	50	50	95.1	169.6	389.7	139.0
rand-2-40-19-fcd	50	50	50	50	50	5,450.4	10,028.1	23,355.0	7,956.0
rand-2-40-19	50	50	50	50	50	10,921.7	20,191.8	47,567.4	15,995.8
rand-2-50-23-fcd	50	12	7	3	10	16,180.0	>27,833.1	>40,818.9	>22,098.5
rand-2-50-23	50	12	3	1	5	25,007.6	>37,142.3	>43,083.2	>33,035.5
rlfapGraphs	14	14	14	14	14	134.1	172.2	145.6	169.3
rlfapGraphsMod	12	12	12	12	12	92.5	144.0	382.5	125.7
rlfapScens11	12	7	5	5	6	3,798.3	>7,784.1	>9,995.0	>5,591.2
rlfapScens	11	11	11	11	11	121.7	156.6	137.4	146.9
rlfapScensMod	13	13	13	13	13	58.8	86.1	164.7	77.3

Table A.1 (continued)

		# Solved				ΣCPU (sec)			
	# Instances	CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
subs	9	9	9	9	9	11.5	15.6	14.0	14.0
super-jobShop-super-jobShop-e0ddr1	10	3	3	2	3	353.9	548.1	>3,660.6	509.0
super-jobShop-super-jobShop-e0ddr2	10	0	0	0	0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-enddr1	10	3	2	2	3	1,439.9	>3,764.2	>3,967.7	1,880.2
super-jobShop-super-jobShop-enddr2	6	2	2	2	2	82.0	168.5	183.4	191.5
super-jobShop-super-jobShop-ewddr2	10	7	7	6	7	1,411.9	2,153.7	>3,884.5	2,678.3
super-os-super-os-taillard-4	30	30	28	27	30	3,382.3	>9,786.8	>16,242.8	5,466.2
super-os-super-os-taillard-5	30	11	10	7	11	4,581.5	>8,461.3	>23,326.7	4,658.5
super-queens	14	5	5	5	5	946.9	1,533.8	2,793.3	1,299.1
tightness0.1	100	100	100	100	100	2,209.7	3,338.2	6,279.2	2,566.3
tightness0.2	100	100	100	100	100	2,394.4	3,861.6	7,892.1	3,116.2
tightness0.35	100	100	100	100	100	1,665.3	3,069.3	7,015.5	2,464.9
tightness0.5	100	100	100	100	100	2,016.1	4,283.2	10,884.4	3,396.1
tightness0.65	100	100	100	100	100	1,437.9	3,426.3	9,574.1	2,674.9
tightness0.8	100	100	100	100	100	1,403.1	3,357.3	10,438.7	2,662.2
tightness0.9	100	100	100	100	100	2,321.4	4,498.5	14,264.9	3,574.4

Table A.2: Aggregated instances results for GAC algorithms for all tested non-binary benchmarks.

	# Instances	# Solved				ΣCPU (sec)			
		CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
QG3	7	7	7	7	7	0.3	0.3	0.3	0.3
QG4	7	7	7	7	7	0.3	0.3	0.3	0.3
QG5	7	0	0	0	0	0.0	0.0	0.0	0.0
QG6	7	7	7	7	7	0.3	0.3	0.3	0.4
QG7	7	7	7	7	7	0.3	0.3	0.3	0.4
aim-100	24	24	24	24	24	17.9	19.0	28.0	21.4
aim-200	24	24	24	24	24	1,831.8	1,989.3	3,142.1	2,323.9
aim-50	24	24	24	24	24	0.6	0.6	0.7	0.7
allIntervalSeries	25	17	16	16	17	2,447.8	>3,625.1	>3,647.1	2,945.6
bddLarge	35	35	35	35	35	2,924.3	7,896.1	5,352.0	2,751.0
bddSmall	35	35	35	35	35	1,423.9	10,676.4	2,000.3	1,403.6
bmc	24	1	1	1	1	70.4	69.6	69.8	70.1
bqwh-15-106_glb	100	100	100	100	100	2.6	2.7	2.6	2.8
bqwh-18-141_glb	100	100	100	100	100	4.3	4.8	4.5	4.7
chessboardColoration	20	13	14	14	14	>4,270.9	1,793.7	2,430.2	1,450.9
dag-half	25	23	2	7	22	13,110.6	>76,820.4	>67,679.4	>23,827.7
dag-rand	25	25	9	25	25	890.5	>85,734.6	2,334.1	1,533.4
dubois	13	6	6	6	6	3,770.9	3,738.6	4,560.8	4,383.3
golombRulerArity3	14	10	9	9	9	1,949.2	>4,527.9	>8,391.8	>4,352.9
golombRulerArity4	14	2	2	2	2	413.7	413.8	413.8	413.7
graceful	4	3	3	2	3	553.7	1,297.1	>3,606.0	1,041.8
jnhSat	16	16	16	16	16	8.0	16.1	14.8	8.3
jnhUnsat	34	34	34	34	34	16.7	32.1	33.2	17.2
latinSquare	10	5	5	5	5	0.6	0.9	0.6	0.7

Table A.2 (continued)

	# Instances	# Solved				ΣCPU (sec)			
		CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
lexVg	63	63	63	63	63	434.1	4,868.3	2,243.9	662.6
mknap	6	2	2	2	2	122.4	122.4	122.4	122.4
modifiedRenault	50	50	50	50	50	51.2	71.0	70.8	61.8
nengfa	10	4	4	4	4	165.6	195.6	245.3	172.8
ogdVg	65	46	40	41	46	5,635.9	>31,226.9	>27,153.2	3,431.5
ortholatin	9	1	1	1	1	0.0	0.0	0.0	0.0
pret	8	4	4	4	4	297.4	294.9	363.7	349.5
primes-10	32	12	12	12	12	7.2	10.2	7.0	7.1
primes-15	32	8	8	8	8	0.9	1.4	0.9	1.0
primes-20	32	8	8	8	8	2.9	5.4	2.9	3.2
primes-25	32	8	8	8	8	4.3	9.5	4.2	4.6
primes-30	32	6	6	6	6	2.4	5.6	2.3	3.4
pseudo-aim	48	48	48	48	48	883.0	958.0	1,430.0	1,071.6
pseudo-chnl	21	1	0	0	1	2,739.4	>3,600.0	>3,600.0	2,896.1
pseudo-circuits	7	3	3	3	3	15.7	16.4	15.0	14.9
pseudo-fpga	36	3	2	2	3	3,169.4	>5,861.6	>5,675.0	3,258.6
pseudo-garden	7	6	7	6	6	>3,600.1	25.4	>3,600.1	>3,600.1
pseudo-ii	41	9	9	9	9	217.2	223.9	571.1	350.7
pseudo-jnh	16	16	16	16	16	9.6	21.4	21.2	9.9
pseudo-logic-synthesis	17	1	1	1	1	26.3	27.1	23.6	25.1
pseudo-mps	49	7	8	7	7	>5,646.0	2,076.6	>5,641.6	>5,644.1
pseudo-mpsReduced	105	1	1	1	1	234.7	235.0	234.7	234.7
pseudo-niklas	19	3	3	3	3	1,743.1	1,743.9	1,738.6	1,741.2
pseudo-par	30	20	20	20	20	520.1	558.2	763.4	549.6
pseudo-primesDimacs	11	4	4	3	3	3,078.1	3,316.6	>3,646.5	>3,646.0

Table A.2 (continued)

	# Instances	# Solved				ΣCPU (sec)			
		CompactTable	GAC2001	STR2	STRbit	CompactTable	GAC2001	STR2	STRbit
pseudo-radar	12	6	6	6	6	126.0	132.1	113.1	120.2
pseudo-routing	15	7	7	7	7	552.7	2,659.6	1,225.9	639.5
pseudo-ssa	8	7	7	7	7	230.2	217.2	294.6	255.7
pseudo-ttp	8	2	2	2	2	0.7	0.8	1.1	0.8
pseudo-uclid	39	8	8	8	8	3,181.8	3,278.4	3,559.6	3,396.8
ramsey3	8	2	2	2	2	0.1	0.1	0.1	0.1
ramsey4	8	1	1	1	1	1.0	0.9	1.0	1.1
rand-10-20-10	20	20	20	20	20	7.8	105.2	20.2	11.4
rand-3-20-20-fcd	50	50	50	50	50	853.6	5,391.0	14,363.0	1,661.3
rand-3-20-20	50	50	50	50	50	1,549.3	10,213.2	28,326.3	3,120.1
rand-3-24-24-fcd	50	41	22	14	39	24,043.0	>92,063.8	>116,608.0	>42,871.2
rand-3-24-24	50	32	8	3	25	34,079.1	>95,905.8	>108,393.4	>55,948.0
rand-3-28-28-fcd	50	8	2	2	5	13,297.4	>22,865.8	>24,247.5	>17,726.8
rand-3-28-28	50	4	1	1	4	3,384.2	>11,386.5	>12,904.6	5,449.9
rand-8-20-5	20	20	19	20	20	374.2	>23,989.3	2,786.2	746.3
renault	2	2	2	2	2	1.9	2.4	1.9	2.3
schurrLemma	10	9	9	9	9	616.0	1,907.3	2,296.7	983.1
small	5	4	4	4	4	213.5	360.4	228.1	347.9
socialGolfers	12	1	1	1	1	>3,999.3	>4,067.5	>3,862.9	>4,102.8
ssa	8	7	7	7	7	148.6	148.5	193.5	170.5
travellingSalesman-20	15	15	15	15	15	110.9	308.6	257.2	128.0
travellingSalesman-25	15	15	15	15	15	1,302.0	3,315.0	3,963.0	1,570.0
ukVg	65	40	34	36	40	9,427.6	>28,733.8	>20,980.2	8,968.5
varDimacs	9	9	9	9	9	1,543.3	2,483.9	2,538.0	1,831.6
wordsVg	65	65	63	65	65	1,386.4	>19,670.8	8,239.4	1,946.9

Appendix B

Per-Benchmark Results for PWC Algorithms

Table B.1 shows detailed results for the PWC algorithms tested in Chapter 4.

Table B.1: Aggregated instance results for PWC algorithms for all tested benchmarks with at least one non-trivial subscope.

		# Solved										ΣCPU (sec)									
	# Instances	PW-AC	PW-AC2^f	PW-AC2-CT	PW-AC2	PW-AC2^f	PW-CT	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2^m	PW-AC	PW-AC2^f	PW-AC2-CT	PW-AC2	PW-CT^f	PW-CT	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2rr
aim-100	24	24	24	24	24	24	24	24	24	24	24	3.8	2.5	3.9	2.8	1.7	2.5	3.1	4.0	4.4	3.4
aim-200	24	24	24	24	24	24	24	24	24	24	24	16.1	15.3	18.8	19.1	9.4	11.3	18.5	48.7	45.2	17.8
aim-50	24	24	24	24	24	24	24	24	24	24	24	1.3	0.9	1.3	1.0	0.6	1.0	0.9	0.9	1.1	1.1
bddLarge	35	0	0	0	0	35	0	0	0	0	0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	3,188.2	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0
bddSmall	35	0	0	0	0	35	35	0	0	0	0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	1,245.5	1,463.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0
bmc	24	1	1	1	1	1	1	1	1	1	1	93.6	80.6	80.6	75.8	76.6	77.0	88.9	86.5	75.7	76.2
chessboardColoration	20	9	11	10	10	12	10	10	10	10	10	>11,424.1	>4,423.5	>7,624.8	>7,441.0	490.3	>7,419.7	>10,139.8	>9,998.6	>7,800.2	>7,786.2
dag-half	25	2	2	7	5	11	15	2	2	3	3	>48,572.7	>47,768.5	>35,168.8	>40,229.5	>33,263.4	24,368.2	>48,712.8	>50,007.3	>47,217.6	>45,562.2
dag-rand	25	25	25	25	25	25	25	25	25	25	25	1,346.2	1,935.0	1,868.0	1,876.0	668.5	675.1	1,327.9	1,315.6	1,283.1	1,295.0
dubois	13	7	7	7	7	9	9	7	6	6	7	>11,552.6	>12,295.3	>10,843.3	>12,292.2	5,871.8	5,885.5	>13,270.9	>15,256.1	>15,300.9	>13,331.0
golombRulerArity4	14	2	2	2	2	2	2	2	2	2	2	414.8	414.4	414.0	414.1	413.7	413.7	414.8	414.6	414.0	414.0
jnhSat	16	16	16	16	16	16	16	16	16	16	16	331.4	220.7	50.4	99.7	17.1	26.0	680.1	497.5	175.6	186.9
jnhUnsat	34	34	34	34	34	34	34	34	34	34	34	814.8	490.9	107.8	217.8	34.2	54.7	1,341.5	1,161.6	372.6	419.9
modifiedRenault	50	50	50	50	50	50	50	50	50	50	50	317.4	191.5	97.7	102.6	69.5	64.1	462.6	411.6	87.6	89.8
nengfa	10	2	4	4	4	4	4	4	4	4	4	>7,317.0	622.5	730.5	372.3	178.9	190.8	2,054.4	1,219.2	316.8	425.0

Table B.1 (continued)

	# Instances	# Solved										ΣCPU (sec)									
		PW-AC	PW-AC2^f	PW-AC2-CT	PW-AC2	PW-AC2^f	PW-CT	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2^m	PW-AC	PW-AC2^f	PW-AC2-CT	PW-AC2	PW-CT^f	PW-CT	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2rr
primes-10	32	12	12	12	12	12	12	12	12	12	12	10.4	10.6	9.5	10.4	7.3	7.3	7.4	7.4	7.4	7.4
primes-15	32	8	8	8	8	8	8	8	8	8	8	1.6	1.6	1.4	1.5	1.0	1.0	1.0	1.0	1.0	1.0
primes-20	32	8	8	8	8	8	8	8	8	8	8	4.7	4.7	4.3	4.6	3.0	3.0	3.2	3.2	3.2	3.2
primes-25	32	8	8	8	8	8	8	8	8	8	8	7.0	7.1	6.4	7.0	4.5	4.5	4.8	4.7	4.7	4.7
primes-30	32	6	6	6	6	6	6	6	6	6	6	3.3	3.5	3.1	3.4	2.4	2.4	2.5	2.5	2.5	2.5
pseudo-aim	48	48	48	48	48	48	48	48	48	48	48	2,663.3	1,809.1	1,981.3	2,288.8	816.8	818.2	3,974.7	3,877.7	3,546.9	3,765.9
pseudo-circuits	7	3	3	3	3	3	3	3	3	3	3	56.7	62.4	26.2	29.2	19.7	19.4	68.1	62.2	27.9	29.4
pseudo-fpga	36	1	2	2	2	2	2	0	1	2	2	>7,147.8	5,730.6	2,829.0	4,514.1	1,678.6	1,680.0	>7,200.0	>7,124.0	4,392.3	5,365.3
pseudo-garden	7	6	7	6	7	6	6	6	6	6	6	>3,600.2	36.6	>3,600.3	41.8	>3,600.2	>3,600.2	>3,600.2	>3,600.2	>3,600.2	>3,600.2
pseudo-ii	41	8	9	9	8	9	9	8	8	8	9	>3,744.8	2,076.8	980.4	>3,664.8	996.2	585.0	>3,705.7	>3,688.7	>3,668.4	3,076.2
pseudo-jnh	16	16	16	16	16	16	16	16	16	16	16	427.1	211.4	47.2	82.6	14.3	21.0	520.9	446.1	175.4	210.2
pseudo-logic-synthesis	17	0	0	1	1	1	1	1	1	1	1	>3,600.0	>3,600.0	55.2	56.7	53.2	46.0	139.3	121.0	42.4	44.9
pseudo-mps	49	6	7	6	7	7	7	6	6	7	7	>9,199.2	>5,632.6	>9,199.1	>5,633.4	>5,781.5	>5,713.5	>9,198.9	>9,198.9	>5,692.0	>5,698.9
pseudo-mpsReduced	105	1	1	1	1	1	1	1	1	1	1	235.0	234.9	234.8	234.9	234.7	234.7	234.7	234.7	234.7	234.7
pseudo-niklas	19	2	2	2	2	3	3	2	2	3	3	>5,299.9	>5,299.3	>5,298.0	>5,297.6	1,887.7	1,812.0	>5,300.4	>5,299.8	1,790.2	1,797.4
pseudo-par	30	20	20	20	20	20	20	20	20	20	20	2,395.1	1,127.0	1,063.5	964.9	538.5	542.7	1,680.0	1,840.4	1,305.2	1,166.0
pseudo-primesDimacs	11	3	3	1	3	3	3	3	3	3	3	44.5	41.8	>7,216.9	44.5	59.3	62.7	66.2	66.0	68.3	68.4

Table B.1 (continued)

	# Instances	# Solved										ΣCPU (sec)									
		PW-AC	PW-AC2^f	PW-AC2-CT	PW-AC2	PW-AC2^f	PW-CT	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2^m	PW-AC	PW-AC2^f	PW-AC2-CT	PW-AC2	PW-CT^f	PW-CT	eSTR2	eSTR2^w	eSTR2^{wm}	eSTR2m
pseudo-ssa	8	7	7	7	7	7	7	7	7	7	7	474.9	284.4	404.7	318.9	241.4	244.7	465.4	477.9	472.6	459.4
pseudo-ttp	8	2	2	2	2	2	2	2	2	2	2	5.2	3.2	2.4	2.0	1.3	1.3	6.2	5.8	2.8	2.6
pseudo-uclid	39	7	9	8	8	9	9	7	8	8	9	>9,393.1	5,313.3	>7,908.5	>6,427.1	4,522.2	4,719.8	>11,631.0	>9,280.9	>7,966.1	6,562.8
rand-10-20-10	20	20	20	20	20	20	20	20	20	20	20	14.1	14.7	14.0	14.1	6.9	7.0	10.0	10.0	9.7	9.8
rand-3-20-20-fcd	50	44	49	50	50	50	50	49	49	50	50	>51,805.8	>28,167.4	8,517.2	16,664.4	1,946.8	1,959.3	>35,709.3	>37,268.7	23,610.2	23,082.7
rand-3-20-20	50	36	46	50	49	50	50	46	45	49	48	>88,116.9	>54,064.6	17,572.5	>35,684.0	3,950.5	3,945.1	>67,747.5	>70,110.4	>48,272.9	>46,253.3
rand-3-24-24-fcd	50	4	5	18	11	37	37	5	5	11	11	>124,213.0	>120,747.5	>88,001.9	>108,554.0	41,039.2	40,262.8	>121,367.7	>121,266.2	>112,492.4	>112,710.2
rand-3-24-24	50	0	1	6	3	24	24	1	1	3	3	>86,400.0	>83,954.6	>72,514.1	>81,504.4	34,752.9	34,506.2	>84,660.7	>85,532.6	>82,387.9	>82,161.0
rand-3-28-28-fcd	50	0	1	2	1	6	6	1	1	2	2	>21,600.0	>20,130.3	>15,673.4	>18,981.0	11,934.8	11,931.2	>20,419.8	>20,383.7	>19,288.8	>19,260.5
rand-3-28-28	50	0	1	1	1	3	3	1	1	1	1	>10,800.0	>9,678.3	>7,842.4	>9,785.6	3,372.2	3,369.4	>10,198.8	>9,954.9	>8,821.5	>8,962.1
rand-8-20-5	20	20	20	20	20	20	20	20	20	20	20	5,765.4	3,860.3	1,872.6	3,005.0	2,423.1	2,054.4	6,421.5	10,985.4	7,447.2	4,144.9
renault	2	2	2	2	2	2	2	2	2	2	2	11.7	7.0	3.8	3.9	2.6	2.6	17.8	14.7	3.1	3.5
schurrLemma	10	6	7	8	8	9	9	7	7	8	8	>12,466.9	>8,616.4	>5,668.7	>5,874.8	1,741.4	1,749.8	>8,958.0	>8,767.4	>5,251.3	>5,579.5
small	5	0	2	2	2	4	4	0	0	3	3	>14,400.0	>7,307.1	>7,351.1	>7,306.4	301.8	307.6	>14,400.0	>14,400.0	>3,862.6	>3,909.7
socialGolfers	12	0	1	0	0	0	0	1	1	1	1	>3,600.0	31.9	>3,600.0	>3,600.0	>3,600.0	>3,600.0	674.4	175.1	151.8	501.0
ssa	8	7	7	7	7	7	7	7	7	7	7	201.2	213.1	257.8	248.8	146.4	147.1	359.4	322.1	305.9	334.0

Appendix C

Per-Benchmark Results for R($*$,m)C Algorithms

Tables C.1, C.2, C.3, C.4, C.5, C.6 show detailed results for the R($*$,m)C algorithms tested in Chapter 5.

Table C.1: Aggregated instances results for AllSol algorithms for all tested binary benchmarks.

	# Instances	# Solved						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
BH-4-13	7	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
BH-4-4	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
BH-4-7	20	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QCP-10	15	0	0	12	12	0	12	>43,200.0	>43,200.0	638.9	625.6	>43,200.0	633.6
QCP-15	15	0	0	1	1	0	1	>3,600.0	>3,600.0	1,012.5	1,011.0	>3,600.0	1,003.7
QCP-20	15	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QCP-25	15	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QWH-10	10	0	0	10	10	0	10	>36,000.0	>36,000.0	129.8	129.9	>36,000.0	126.0
QWH-15	10	0	0	4	4	0	4	>14,400.0	>14,400.0	3,114.6	3,119.0	>14,400.0	3,089.7
QWH-20	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QWH-25	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bqwh-15-106	100	1	1	100	100	1	100	>356,915.8	>356,950.9	11,987.7	12,300.5	>356,910.8	11,188.4
bqwh-18-141	100	0	0	62	62	0	63	>226,800.0	>226,800.0	>76,302.3	>77,646.6	>226,800.0	71,895.4
coloring	22	17	17	21	21	17	21	>14,753.5	>14,772.4	1,287.5	1,259.2	>14,758.9	1,308.6
composed-25-1-2	10	10	10	10	10	10	10	2.1	2.1	3.3	3.3	1.9	3.2
composed-25-1-25	10	10	10	10	10	10	10	2.7	2.7	4.9	4.9	2.4	4.7

Table C.1 (continued)

		# Instances completed						ΣCPU (sec)					
	# Instances	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
composed-25-1-40	10	10	10	10	10	10	10	24.4	25.6	5.8	5.7	24.3	5.6
composed-25-1-80	10	9	9	10	10	9	10	>5,096.8	>5,182.2	23.0	21.5	>5,104.9	24.9
composed-25-10-20	10	0	0	5	5	0	5	>18,000.0	>18,000.0	258.4	249.2	>18,000.0	268.7
composed-75-1-2	10	10	10	10	10	10	10	7.7	7.7	24.3	23.7	7.4	25.6
composed-75-1-25	10	10	10	10	10	10	10	8.4	8.4	28.3	27.5	8.0	29.8
composed-75-1-40	10	10	10	10	10	10	10	306.4	292.9	54.2	52.0	295.4	58.8
composed-75-1-80	10	1	2	10	10	1	10	>32,459.9	>31,310.7	62.2	59.3	>32,456.4	67.4
domino	24	7	7	7	7	7	6	6,015.7	6,015.8	6,015.7	6,015.9	6,011.1	>7,873.2
driver	7	1	1	4	4	1	4	>10,800.1	>10,800.1	3,897.6	3,727.7	>10,800.1	3,813.3
ehi-85	100	1	1	50	50	1	50	>176,527.7	>176,539.6	2,003.2	1,960.3	>176,531.0	2,115.9
ehi-90	100	0	0	54	54	0	54	>194,400.0	>194,400.0	2,662.2	2,598.6	>194,400.0	2,802.4
fapp-fapp01	11	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb30-15	10	0	0	10	10	0	10	>36,000.0	>36,000.0	10,874.4	10,145.5	>36,000.0	11,091.6
frb35-17	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb40-19	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb45-21	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.1 (continued)

	# Instances	# Instances completed						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
frb50-23	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb53-24	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb56-25	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb59-26	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
geom	100	0	0	72	72	0	72	>259,200.0	>259,200.0	26,030.9	23,403.1	>259,200.0	29,355.4
graphColoring-hos	14	1	1	6	6	1	6	>18,369.8	>18,398.8	1,611.0	1,560.1	>18,371.0	1,694.7
graphColoring-insertion-full-insertion	41	5	5	16	16	5	17	>43,609.8	>43,567.5	>7,732.9	>7,607.7	>43,594.5	7,766.2
graphColoring-insertion-k-insertion	32	7	7	15	15	7	15	>29,843.1	>29,823.0	4,327.8	4,420.7	>29,811.3	4,145.2
graphColoring-leighton-leighton-15	28	4	4	2	2	6	2	>7,653.1	>7,651.9	>18,471.9	>18,228.3	1,348.4	>18,855.2
graphColoring-leighton-leighton-25	32	2	2	2	2	2	2	157.7	157.7	2,450.4	2,309.3	164.5	2,666.8
graphColoring-leighton-leighton-5	8	4	4	7	7	4	7	>11,138.2	>11,138.8	5,106.7	4,953.8	>11,151.6	5,288.9
graphColoring-mug	8	8	8	4	4	8	4	347.9	352.1	>14,400.4	>14,400.4	341.9	>14,400.4
graphColoring-myciel	16	5	5	10	10	5	10	>18,223.5	>18,240.4	2,906.5	2,927.5	>18,222.6	2,787.5
graphColoring-register-fpsol	37	3	3	2	2	3	2	255.8	255.8	>5,563.2	>5,481.3	270.5	>5,657.6
graphColoring-register-inithx	32	3	4	2	2	5	2	>10,012.0	>9,283.7	>13,773.9	>13,662.5	5,850.8	>13,924.0
graphColoring-register-mulsol	49	9	9	5	5	9	5	10,185.5	10,621.2	>17,731.5	>17,615.1	10,406.9	>17,848.4

Table C.1 (continued)

		# Instances completed						ΣCPU (sec)					
	# Instances	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
graphColoring-register-zeroin	31	5	5	3	3	5	3	4,975.4	5,198.8	>8,467.8	>8,417.4	5,070.7	>8,537.6
graphColoring-school	8	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
graphColoring-sgb-book	26	19	19	10	10	19	10	>19,309.4	>19,578.0	>47,225.4	>47,198.0	>19,353.3	>47,257.4
graphColoring-sgb-games	4	3	3	2	2	3	2	>3,702.8	>3,708.9	>9,811.7	>9,812.5	>3,702.5	>9,746.2
graphColoring-sgb-miles	42	7	7	7	7	7	7	>5,307.6	>5,386.8	>6,736.6	>6,476.5	>5,335.2	>7,196.5
graphColoring-sgb-queen	50	3	3	6	6	3	6	>14,475.9	>14,481.1	>8,178.1	>8,090.1	>14,476.4	>8,219.4
hanoi	5	5	4	5	5	5	4	2,330.1	>3,607.5	2,330.1	2,330.2	2,324.5	>3,607.4
haystacks	51	7	7	3	3	7	3	1,435.7	1,517.4	>14,444.1	>14,445.1	1,448.4	>14,441.0
jobShop-e0ddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-e0ddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-enddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-enddr2	6	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-ewddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
knights	19	6	6	7	7	6	7	>3,790.5	>3,790.9	2,879.7	1,797.6	>3,797.4	3,328.3
langford	4	4	4	2	2	4	2	2,744.0	2,837.6	>7,438.2	>7,405.5	2,824.0	>7,467.6
langford2	24	9	9	14	14	9	14	>21,914.7	>21,928.8	>8,878.7	>8,060.8	>21,922.7	>10,285.1

Table C.1 (continued)

	# Instances	# Instances completed						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
langford3	23	10	10	9	9	9	9	1,587.2	1,641.7	>4,948.3	>4,766.0	>3,833.8	>5,091.0
langford4	24	10	10	8	8	10	8	1,185.9	1,218.2	>10,069.1	>9,587.0	1,217.8	>10,613.7
lard	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
marc	10	7	7	10	10	7	8	>11,619.5	>11,619.5	13,111.5	10,911.7	>11,593.7	>16,338.2
os-taillard-4	30	4	4	19	19	4	15	>62,519.6	>62,598.8	>24,484.8	>23,130.3	>62,844.5	>36,281.1
os-taillard-5	30	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
os-taillard-7	30	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pigeons	25	9	9	11	11	9	11	>11,790.0	>11,843.1	>5,046.3	>4,985.4	>11,802.1	>5,158.1
queenAttacking	10	2	2	2	2	2	2	4.3	4.5	52.0	47.3	4.4	57.9
queens	14	4	4	7	7	4	7	>10,878.0	>10,882.1	712.6	607.4	>10,879.3	948.4
queensKnights	18	5	5	7	7	5	7	>12,601.2	>12,708.5	>5,673.8	>5,040.1	>12,778.9	>5,973.6
rand-2-23	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-24	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-25	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-26	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-27	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.1 (continued)

	# Instances	# Instances completed						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
rand-2-30-15-fcd	50	0	0	46	46	0	46	>165,600.0	>165,600.0	52,848.9	49,427.5	>165,600.0	54,035.2
rand-2-30-15	50	0	0	38	39	0	37	>140,400.0	>140,400.0	>66,254.9	62,129.2	>140,400.0	>67,538.0
rand-2-40-19-fcd	50	0	0	1	1	0	1	>3,600.0	>3,600.0	932.9	870.8	>3,600.0	970.0
rand-2-40-19	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-50-23-fcd	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-50-23	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rlfapGraphs	14	5	5	9	9	5	9	>18,858.8	>17,457.0	6,208.3	5,425.5	>18,875.4	7,899.3
rlfapGraphsMod	12	6	6	5	5	6	5	>4,918.4	>4,969.3	>7,658.8	>7,618.4	>4,892.3	>7,690.6
rlfapScens11	12	5	5	2	2	5	2	1,761.2	1,811.0	>12,001.5	>11,874.0	1,788.1	>12,264.4
rlfapScens	11	7	7	8	8	7	8	>4,541.2	>4,570.9	2,865.6	2,542.1	>4,562.9	3,601.7
rlfapScensMod	13	7	7	9	9	7	8	>7,683.9	>7,690.2	3,935.9	3,506.8	>7,682.4	>4,004.9
subs	9	9	9	9	9	9	9	20.6	20.7	209.9	179.5	19.4	272.6
super-jobShop-super-jobShop-e0ddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-e0ddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-enddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-enddr2	6	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.1 (continued)

	# Instances	# Instances completed						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
super-jobShop-super-jobShop-ewddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-os-super-os-taillard-4	30	3	3	12	13	3	9	>41,698.2	>41,811.6	>18,270.1	16,905.9	>42,031.0	>27,312.3
super-os-super-os-taillard-5	30	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-queens	14	4	4	3	3	4	3	178.9	188.0	>3,775.1	>3,768.6	181.9	>3,775.1
tightness0.1	100	0	0	6	6	0	5	>21,600.0	>21,600.0	12,130.3	10,978.7	>21,600.0	>12,552.7
tightness0.2	100	0	0	5	7	0	5	>25,200.0	>25,200.0	>15,822.2	15,088.3	>25,200.0	>15,790.7
tightness0.35	100	0	0	10	13	0	10	>46,800.0	>46,800.0	>24,325.8	23,545.1	>46,800.0	>24,446.4
tightness0.5	100	0	0	13	13	0	13	>46,800.0	>46,800.0	23,784.4	22,669.6	>46,800.0	24,628.8
tightness0.65	100	0	0	33	39	0	30	>140,400.0	>140,400.0	>74,684.1	71,495.0	>140,400.0	>76,991.4
tightness0.8	100	17	20	64	64	17	62	>198,615.5	>186,915.2	73,348.6	68,902.0	>199,513.5	>78,522.3
tightness0.9	100	73	76	79	81	73	79	>97,407.5	>87,323.7	>74,478.6	>71,008.2	>105,932.0	>78,987.1

Table C.2: Aggregated instances results for AllSol algorithms for all tested non-binary benchmarks.

		# Solved						ΣCPU (sec)					
	# Instances	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
QG3	7	7	7	7	7	7	7	0.5	0.5	0.5	0.5	0.5	0.5
QG4	7	7	7	7	7	7	7	0.5	0.5	0.5	0.5	0.5	0.5
QG5	7	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QG6	7	7	7	7	7	7	7	0.6	0.6	0.6	0.6	0.6	0.6
QG7	7	7	7	7	7	7	7	0.6	0.6	0.6	0.6	0.6	0.6
aim-100	24	7	6	18	18	7	18	>47,210.6	>47,647.6	9,530.5	9,883.4	>46,800.5	8,626.2
aim-200	24	0	0	3	3	0	3	>10,800.0	>10,800.0	16.9	17.0	>10,800.0	17.3
aim-50	24	24	24	24	24	24	24	306.2	339.1	72.4	75.1	300.8	65.3
allIntervalSeries	25	6	6	9	9	6	9	>11,824.4	>11,887.2	4,211.1	4,046.8	>11,822.6	4,051.1
bddLarge	35	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bddSmall	35	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bmc	24	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bqwh-15-106_glb	100	34	35	100	100	34	100	>261,863.6	>261,331.5	70.8	61.1	>260,525.0	73.1
bqwh-18-141_glb	100	1	1	100	100	1	100	>356,509.3	>356,524.1	385.1	293.2	>356,504.1	425.2
chessboardColoration	20	6	6	6	7	6	6	>10,950.5	>10,957.7	>11,300.6	>10,514.2	>10,953.6	>11,345.3
dag-half	25	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.2 (continued)

	# Instances	# Instances completed						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
dag-rand	25	22	22	22	22	22	22	1,561.7	1,562.1	1,584.1	1,589.1	870.9	874.8
dubois	13	3	3	3	3	3	3	4,231.4	4,257.6	4,232.0	4,257.9	4,074.5	4,074.0
golombRulerArity3	14	0	0	2	2	0	2	>7,200.0	>7,200.0	763.6	672.8	>7,200.0	886.4
golombRulerArity4	14	2	2	2	2	2	2	414.4	414.4	414.6	414.6	414.0	414.3
graceful	4	1	1	1	1	1	1	67.9	72.5	13.0	12.6	67.7	12.5
jnhSat	16	0	0	0	3	0	0	>10,800.0	>10,800.0	>10,800.0	7,955.7	>10,800.0	>10,800.0
jnhUnsat	34	0	0	5	10	0	4	>36,000.0	>36,000.0	>24,467.6	11,196.5	>36,000.0	>26,732.5
latinSquare	10	4	4	4	5	4	4	>3,607.6	>3,607.9	>3,791.8	2,571.2	>3,607.4	>4,452.8
lexVg	63	37	35	39	39	37	39	>30,430.7	>31,522.4	>19,577.2	>18,204.2	>30,310.7	>23,874.0
mknap	6	2	2	2	2	2	2	122.4	122.4	122.4	122.4	122.4	122.4
modifiedRenault	50	50	50	50	50	50	50	1,862.2	1,885.6	2,848.5	2,738.6	1,673.8	2,754.4
nengfa	10	1	1	2	2	1	2	>3,737.1	>3,741.5	437.4	379.0	>3,733.8	497.1
ogdVg	65	8	8	14	14	8	10	>29,029.7	>29,047.0	>16,918.5	>14,498.3	>29,025.7	>26,147.4
ortholatin	9	1	1	1	1	1	1	0.0	0.0	0.0	0.0	0.0	0.0
pret	8	4	4	4	4	4	4	7,607.6	7,669.2	7,605.3	7,672.7	7,382.6	7,382.7
primes-10	32	12	12	12	12	12	12	36.5	35.7	36.7	35.7	28.4	33.0

Table C.2 (continued)

		# Instances completed						ΣCPU (sec)					
	# Instances	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
primes-15	32	8	8	8	8	8	8	5.6	5.7	5.8	5.8	5.0	5.1
primes-20	32	8	8	8	8	8	8	14.7	14.7	15.8	15.8	12.6	13.7
primes-25	32	8	8	8	8	8	8	24.6	24.1	25.3	24.8	21.7	22.3
primes-30	32	6	6	6	6	6	6	9.8	9.8	9.8	9.8	9.1	9.1
pseudo-aim	48	11	11	25	25	11	25	>56,171.3	>56,459.9	2,984.4	3,040.1	>56,033.1	2,821.2
pseudo-chnl	21	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-circuits	7	3	3	2	2	3	3	182.6	195.0	>3,600.2	>3,600.2	124.2	542.3
pseudo-fpga	36	12	12	12	12	12	12	3,729.6	4,048.3	4,464.0	4,583.8	3,672.6	4,246.8
pseudo-garden	7	6	6	6	6	6	6	4.3	4.7	3.5	3.4	4.0	3.3
pseudo-ii	41	1	1	1	1	1	1	0.4	0.4	0.5	0.5	0.4	0.5
pseudo-jnh	16	0	0	2	5	0	2	>18,000.0	>18,000.0	>12,969.3	9,065.5	>18,000.0	>13,735.8
pseudo-logic-synthesis	17	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-mps	49	6	6	6	6	6	6	2,002.6	2,002.7	2,009.0	2,009.9	2,001.6	2,006.5
pseudo-mpsReduced	105	1	1	1	1	1	1	238.0	238.2	244.3	245.2	237.1	242.0
pseudo-niklas	19	2	2	2	2	2	2	1,736.8	1,741.0	1,800.7	1,771.6	1,731.5	1,795.2
pseudo-par	30	12	12	11	11	12	11	3,634.1	3,751.5	>6,685.0	>6,777.4	3,602.4	>6,445.7

Table C.2 (continued)

	# Instances	# Instances completed						ΣCPU (sec)					
		AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
pseudo-primesDimacs	11	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-radar	12	3	3	3	3	3	2	735.4	717.4	4,384.8	795.4	660.3	>6,899.2
pseudo-routing	15	4	4	4	4	4	4	3,365.1	3,626.6	4,066.4	4,032.7	3,061.2	3,371.6
pseudo-ssa	8	1	1	1	1	1	1	382.1	383.9	388.9	390.6	382.1	386.8
pseudo-ttp	8	2	2	2	2	2	2	1,293.0	1,453.8	3,001.6	3,053.5	1,300.2	2,691.3
pseudo-uclid	39	3	3	3	3	3	3	1,831.0	1,833.6	1,832.9	1,833.7	1,827.5	1,828.9
ramsey3	8	0	0	2	2	0	2	>7,200.0	>7,200.0	6.3	5.8	>7,200.0	6.5
ramsey4	8	0	0	3	3	0	3	>10,800.0	>10,800.0	1,243.8	1,039.1	>10,800.0	1,449.1
rand-10-20-10	20	20	20	20	20	20	20	44.8	44.8	45.1	45.1	40.7	40.9
rand-3-20-20-fcd	50	0	0	4	5	0	3	>18,000.0	>18,000.0	>11,161.9	7,494.3	>18,000.0	>14,259.7
rand-3-20-20	50	0	0	3	3	0	1	>10,800.0	>10,800.0	6,248.0	4,099.4	>10,800.0	>8,001.3
rand-3-24-24-fcd	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-24-24	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-28-28-fcd	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-28-28	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-8-20-5	20	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.2 (continued)

		# Instances completed						ΣCPU (sec)					
	# Instances	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}	AllSolFB$^{\Psi}$-r	AllSolFB$^{\Psi}$	AllSolFB^{m3}-r	AllSolFB^{m3}	AllSol$^{\Psi}$	AllSol^{m3}
renault	2	2	2	2	2	2	2	13.9	14.4	17.7	18.5	8.9	12.6
schurrLemma	10	6	6	4	4	6	3	3,504.5	3,547.9	>12,119.5	>11,550.7	3,529.8	>12,587.0
small	5	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
socialGolfers	12	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
ssa	8	6	6	6	6	6	6	42.7	42.8	44.9	45.0	43.5	45.4
travellingSalesman-20	15	0	0	7	7	0	7	>25,200.0	>25,200.0	4,195.8	4,001.7	>25,200.0	4,353.6
travellingSalesman-25	15	0	0	3	3	0	2	>10,800.0	>10,800.0	8,342.3	7,792.5	>10,800.0	>8,763.9
ukVg	65	8	7	10	11	8	9	>35,705.5	>36,516.1	>30,646.6	>28,516.2	>35,675.1	>38,154.4
varDimacs	9	6	6	6	6	6	6	238.3	262.3	273.3	285.9	232.8	240.6
wordsVg	65	30	30	36	36	30	35	>33,201.1	>34,773.6	>12,047.6	>10,265.0	>33,116.1	>17,382.8

Table C.3: Aggregated instances results for PerTuple algorithms for all tested binary benchmarks.

		# Solved						ΣCPU (sec)					
	# Instances	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
BH-4-13	7	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
BH-4-4	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
BH-4-7	20	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QCP-10	15	0	0	0	12	12	12	>43,200.0	>43,200.0	>43,200.0	431.0	835.5	443.5
QCP-15	15	0	0	0	1	1	1	>3,600.0	>3,600.0	>3,600.0	865.4	1,233.4	861.3
QCP-20	15	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QCP-25	15	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QWH-10	10	7	7	7	10	10	10	>17,247.1	>19,830.2	>17,418.2	113.7	164.5	115.7
QWH-15	10	0	0	0	4	4	4	>14,400.0	>14,400.0	>14,400.0	2,684.6	3,770.5	2,661.8
QWH-20	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QWH-25	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bqwh-15-106	100	29	25	30	100	100	100	>278,561.2	>284,931.8	>280,799.1	9,310.6	17,177.9	9,179.3
bqwh-18-141	100	0	0	0	69	56	67	>248,400.0	>248,400.0	>248,400.0	75,171.2	>125,904.1	>74,852.9
coloring	22	18	18	18	18	21	18	>10,950.5	>10,961.0	>10,953.5	>10,896.6	1,464.8	>10,899.2
composed-25-1-2	10	10	10	10	10	10	10	1.9	1.9	1.9	2.9	2.9	2.9

Table C.3 (continued)

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
composed-25-1-25	10	10	10	10	10	10	10	2.6	2.6	2.6	4.3	4.4	4.3
composed-25-1-40	10	10	10	10	10	10	10	3.3	3.3	3.3	5.1	5.2	5.0
composed-25-1-80	10	10	10	10	10	10	10	23.4	23.3	23.4	18.2	18.7	18.0
composed-25-10-20	10	0	0	0	5	5	5	>18,000.0	>18,000.0	>18,000.0	173.0	296.7	177.3
composed-75-1-2	10	10	10	10	10	10	10	7.2	7.2	7.2	21.0	18.0	20.8
composed-75-1-25	10	10	10	10	10	10	10	7.8	7.8	7.8	24.4	20.8	24.1
composed-75-1-40	10	10	10	10	10	10	10	8.5	8.5	8.5	46.5	37.0	45.8
composed-75-1-80	10	10	10	10	10	10	10	24.3	23.9	24.2	54.5	44.5	53.6
domino	24	7	7	7	7	7	7	6,018.3	6,018.4	6,019.4	6,018.3	6,018.4	6,018.5
driver	7	2	1	1	1	4	1	>10,152.4	>10,800.1	>10,800.1	>10,800.3	5,485.6	>10,800.3
ehi-85	100	8	6	8	50	50	50	>167,899.1	>173,101.4	>167,898.2	>12,975.4	>12,703.9	>12,956.6
ehi-90	100	7	7	7	54	54	54	>194,042.0	>195,401.0	>194,040.9	>17,155.4	>17,028.5	>17,142.8
fapp-fapp01	11	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb30-15	10	0	0	0	10	10	10	>36,000.0	>36,000.0	>36,000.0	7,388.7	17,129.9	7,316.5
frb35-17	10	0	0	0	0	0	0	>7,200.0	>7,200.0	>7,200.0	>7,200.0	>7,200.0	>7,200.0
frb40-19	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.3 (continued)

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
frb45-21	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb50-23	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb53-24	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb56-25	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
frb59-26	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
geom	100	30	24	27	77	71	77	>238,539.3	>258,814.2	>245,679.0	>45,017.6	>68,631.8	>45,326.8
graphColoring-hos	14	2	2	2	4	6	4	>14,899.2	>14,899.1	>14,899.3	>7,324.5	1,804.2	>7,324.9
graphColoring-insertion-full-insertion	41	12	12	12	15	16	15	>30,550.0	>30,991.2	>30,664.6	>22,844.1	>18,877.1	>22,879.9
graphColoring-insertion-k-insertion	32	10	10	10	15	14	15	>18,647.8	>18,693.8	>18,665.8	3,715.2	>4,981.3	3,730.4
graphColoring-leighton-leighton-15	28	6	6	6	0	2	0	>6,522.1	>6,521.9	>6,522.0	>25,200.0	>22,163.2	>25,200.0
graphColoring-leighton-leighton-25	32	4	4	4	0	2	0	4,687.7	4,687.5	4,687.7	>14,400.0	>9,792.9	>14,400.0
graphColoring-leighton-leighton-5	8	4	4	4	0	7	0	>11,144.5	>11,144.5	>11,144.5	>25,200.0	5,446.9	>25,200.0
graphColoring-mug	8	8	8	8	4	4	4	351.6	356.5	352.0	>14,400.4	>14,400.4	>14,400.4
graphColoring-myciel	16	8	8	7	10	10	10	>9,925.4	>9,629.1	>11,910.8	2,351.7	4,122.5	2,516.1
graphColoring-register-fpsol	37	3	3	3	0	2	0	269.2	269.2	269.2	>10,800.0	>5,793.7	>10,800.0
graphColoring-register-inithx	32	3	3	3	0	2	0	>7,607.9	>7,607.9	>7,607.9	>18,000.0	>14,279.2	>18,000.0

Table C.3 (continued)

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
graphColoring-register-mulsol	49	5	5	5	0	5	0	>14,490.6	>14,490.6	>14,490.6	>32,400.0	>18,713.1	>32,400.0
graphColoring-register-zeroin	31	5	5	5	0	3	0	4,988.2	4,988.2	4,988.2	>18,000.0	>8,798.3	>18,000.0
graphColoring-school	8	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
graphColoring-sgb-book	26	23	23	23	10	10	10	>7,756.6	>7,769.5	>7,763.1	>50,603.9	>50,859.1	>50,617.4
graphColoring-sgb-games	4	4	4	4	2	2	2	97.1	129.4	107.1	>8,771.5	>10,446.0	>8,814.2
graphColoring-sgb-miles	42	7	7	7	6	7	6	>5,321.9	>5,322.6	>5,322.3	>8,196.6	>7,360.0	>8,255.7
graphColoring-sgb-queen	50	4	4	4	7	5	7	>11,000.9	>11,049.9	>11,001.9	5,590.2	>8,887.1	5,609.5
hanoi	5	4	5	5	5	5	5	>3,615.3	2,461.3	2,461.5	2,461.5	2,461.4	2,461.5
haystacks	51	8	8	8	3	3	3	3,194.2	3,258.9	3,194.2	>18,048.6	>18,065.0	>18,051.3
jobShop-e0ddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-e0ddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-enddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-enddr2	6	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-ewddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
knights	19	6	6	6	4	4	4	>3,800.1	>3,800.1	>3,800.1	>11,469.8	>11,467.7	>11,469.2
langford	4	2	3	4	2	2	2	>7,260.7	>5,855.9	4,450.1	>7,320.2	>7,490.0	>7,323.8

Table C.3 (continued)

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
langford2	24	10	10	10	15	14	15	>20,084.7	>20,217.7	>20,109.5	4,713.4	>9,222.7	4,706.6
langford3	23	10	10	10	9	9	9	2,607.3	2,614.9	2,607.3	>4,272.1	>5,296.9	>4,283.4
langford4	24	10	9	9	9	8	9	1,919.4	>4,023.1	>4,023.1	>7,148.9	>10,418.8	>7,150.9
lard	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
marc	10	5	5	5	5	10	5	>18,504.3	>18,504.3	>18,504.3	>18,326.1	9,960.3	>18,326.1
os-taillard-4	30	1	1	1	12	8	12	>73,687.8	>73,687.8	>73,687.8	>47,620.4	>55,519.3	>47,973.2
os-taillard-5	30	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
os-taillard-7	30	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pigeons	25	9	9	9	11	11	11	>11,851.4	>11,851.4	>11,851.4	>5,137.9	>5,569.1	>5,149.5
queenAttacking	10	2	2	2	2	2	2	5.1	5.1	5.1	36.6	69.2	36.3
queens	14	5	5	5	7	7	7	>11,336.6	>11,474.3	>11,389.2	>3,779.9	>4,321.4	>3,793.6
queensKnights	18	6	6	6	5	4	5	>9,707.6	>9,845.0	>9,760.3	>13,927.1	>15,074.9	>13,866.2
rand-2-23	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-24	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-25	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-26	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.3 (continued)

		# Solved						ΣCPU (sec)					
	# Instances	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
rand-2-27	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-30-15-fcd	50	2	1	1	48	37	48	>170,212.5	>172,211.6	>170,811.2	41,864.1	>81,533.2	41,273.3
rand-2-30-15	50	1	0	0	44	26	44	>161,764.9	>162,000.0	>162,000.0	>64,771.8	>114,227.2	>64,086.6
rand-2-40-19-fcd	50	0	0	0	2	1	2	>7,200.0	>7,200.0	>7,200.0	3,377.0	>5,044.4	3,213.5
rand-2-40-19	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-50-23-fcd	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-50-23	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rlfapGraphs	14	7	6	7	7	7	7	>10,939.6	>12,445.8	>11,175.1	>8,465.6	>10,345.1	>8,506.5
rlfapGraphsMod	12	8	8	8	5	5	5	>3,837.1	>3,851.2	>3,836.3	>14,685.6	>14,988.1	>14,691.5
rlfapScens11	12	5	5	5	0	2	0	255.2	253.9	254.6	>18,000.0	>12,042.6	>18,000.0
rlfapScens	11	7	7	7	7	7	7	>4,229.4	>4,323.2	>4,231.0	>4,461.1	>4,641.3	>4,462.7
rlfapScensMod	13	9	9	9	8	8	8	>4,567.8	>5,178.5	>4,581.9	>7,523.7	>7,693.3	>7,528.3
subs	9	9	9	9	9	9	9	21.5	21.9	21.5	277.7	350.9	281.0
super-jobShop-super-jobShop-e0ddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-e0ddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-enddr1	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.3 (continued)

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
super-jobShop-super-jobShop-enddr2	6	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-ewddr2	10	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-os-super-os-taillard-4	30	2	2	2	4	4	4	>43,148.3	>43,148.3	>43,148.3	>36,036.4	>37,071.7	>36,039.0
super-os-super-os-taillard-5	30	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
super-queens	14	4	4	4	3	3	3	293.7	293.7	293.7	>3,738.8	>3,870.9	>3,756.2
tightness0.1	100	0	0	0	8	5	8	>36,000.0	>36,000.0	>36,000.0	>19,135.2	>29,134.4	>19,175.0
tightness0.2	100	0	0	0	8	3	8	>28,800.0	>28,800.0	>28,800.0	13,494.3	>22,545.6	13,209.3
tightness0.35	100	0	0	0	22	9	20	>90,000.0	>90,000.0	>90,000.0	>57,117.4	>74,219.9	>56,795.4
tightness0.5	100	0	0	0	17	7	18	>72,000.0	>72,000.0	>72,000.0	>42,416.7	>61,234.8	>40,449.5
tightness0.65	100	12	7	10	34	19	39	>172,655.0	>183,770.8	>173,823.6	>124,462.0	>157,460.4	>120,086.3
tightness0.8	100	67	59	66	58	39	59	>103,364.1	>132,386.3	>104,315.4	>133,804.1	>193,606.9	>129,158.4
tightness0.9	100	77	72	79	66	50	67	>98,575.9	>126,323.3	>95,561.8	>143,477.3	>195,727.5	>141,663.6

Table C.4: Aggregated instances results for PerTuple algorithms for all tested non-binary benchmarks.

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
QG3	7	7	7	7	7	7	7	0.5	0.5	0.5	0.5	0.5	0.5
QG4	7	7	7	7	7	7	7	0.5	0.5	0.5	0.5	0.5	0.5
QG5	7	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
QG6	7	7	7	7	7	7	7	0.6	0.6	0.6	0.6	0.6	0.6
QG7	7	7	7	7	7	7	7	0.6	0.6	0.6	0.6	0.6	0.6
aim-100	24	15	15	15	18	17	18	>18,789.9	>22,528.0	>18,882.3	9,540.8	>12,122.8	9,487.2
aim-200	24	2	2	2	3	3	3	>3,613.6	>3,616.1	>3,614.8	18.2	17.8	18.4
aim-50	24	24	24	24	24	24	24	104.4	133.3	121.9	74.9	96.9	76.0
allIntervalSeries	25	7	7	7	9	8	9	>7,409.3	>7,465.3	>7,417.2	3,232.3	>5,162.5	3,184.5
bddLarge	35	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bddSmall	35	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bmc	24	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
bqwh-15-106_glb	100	98	98	98	100	100	100	>9,199.5	>11,237.1	>10,144.0	74.9	92.3	73.8
bqwh-18-141_glb	100	54	53	55	100	100	100	>187,567.0	>184,130.9	>180,876.1	377.9	531.8	348.7
chessboardColoration	20	8	8	8	7	6	7	>3,743.5	>3,786.7	>3,766.1	>9,062.5	>11,385.6	>9,025.2

Table C.4 (continued)

		# Solved						ΣCPU (sec)					
	# Instances	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
dag-half	25	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
dag-rand	25	22	22	22	22	22	22	873.5	873.1	873.5	915.8	909.1	915.3
dubois	13	3	3	3	3	3	3	4,082.6	4,116.6	4,063.7	4,082.8	4,118.2	4,065.5
golombRulerArity3	14	0	0	0	2	2	2	>7,200.0	>7,200.0	>7,200.0	339.3	892.7	357.2
golombRulerArity4	14	2	2	2	2	2	2	414.1	414.1	414.1	415.2	415.0	415.2
graceful	4	1	1	1	1	1	1	3.3	4.2	3.7	10.9	19.7	10.8
jnhSat	16	0	0	0	0	0	0	>32,400.0	>32,400.0	>32,400.0	>32,400.0	>32,400.0	>32,400.0
jnhUnsat	34	0	0	0	1	6	1	>61,200.0	>61,200.0	>61,200.0	>57,826.0	>46,186.3	>57,823.5
latinSquare	10	4	4	4	4	4	4	>3,601.6	>3,601.6	>3,601.6	>3,758.2	>3,836.1	>3,781.9
lexVg	63	47	47	47	39	36	39	9,746.1	10,911.4	9,782.7	>38,086.3	>49,216.9	>36,761.8
mknap	6	2	2	2	2	2	2	122.4	122.4	122.4	122.4	122.4	122.4
modifiedRenault	50	50	50	50	50	50	50	2,572.0	3,278.5	2,535.0	11,357.2	18,520.8	11,129.3
nengfa	10	1	1	1	2	2	2	>3,716.4	>3,721.9	>3,715.8	527.8	865.9	527.7
ogdVg	65	18	18	19	11	10	11	>9,702.3	>11,531.9	8,193.1	>36,060.8	>39,722.2	>36,073.3
ortholatin	9	1	1	1	1	1	1	0.0	0.0	0.0	0.0	0.0	0.0
pret	8	4	4	4	4	4	4	7,297.5	7,490.6	7,279.4	7,297.0	7,491.7	7,278.2

Table C.4 (continued)

	# Instances	# Solved						ΣCPU (sec)					
		PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
primes-10	32	12	12	12	12	12	12	964.6	966.3	964.1	1,059.8	1,065.2	1,059.5
primes-15	32	8	8	8	8	8	8	10.3	10.3	10.3	10.4	10.4	10.4
primes-20	32	8	8	8	8	8	8	89.6	89.5	89.6	90.6	91.0	90.6
primes-25	32	8	8	8	8	8	8	122.8	122.5	122.7	123.3	123.9	123.2
primes-30	32	6	6	6	6	6	6	252.1	252.0	252.1	252.1	252.0	252.1
pseudo-aim	48	21	20	21	25	25	25	>22,372.8	>24,301.7	>23,201.4	2,895.2	3,423.2	2,886.6
pseudo-chnl	21	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-circuits	7	2	2	2	2	2	2	>3,600.1	>3,600.2	>3,600.1	>3,600.6	>3,600.9	>3,600.6
pseudo-fpga	36	12	11	13	11	9	11	>4,922.5	>14,143.5	3,159.4	>13,489.5	>20,219.7	>13,435.4
pseudo-garden	7	6	6	6	6	6	6	2.3	3.3	2.6	3.6	4.2	3.6
pseudo-ii	41	1	1	1	1	1	1	0.4	0.5	0.4	0.6	0.7	0.6
pseudo-jnh	16	0	0	0	0	2	0	>32,400.0	>32,400.0	>32,400.0	>32,400.0	>27,214.1	>32,400.0
pseudo-logic-synthesis	17	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-mps	49	6	6	6	6	6	6	2,031.5	2,044.2	2,031.2	2,042.2	2,056.3	2,041.9
pseudo-mpsReduced	105	1	1	1	1	1	1	251.4	264.0	251.2	262.0	276.0	261.7
pseudo-niklas	19	2	2	2	1	1	1	2,925.7	3,309.6	2,868.7	>5,309.6	>5,309.6	>5,309.6

Table C.4 (continued)

		# Solved						ΣCPU (sec)					
	# Instances	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
pseudo-par	30	12	12	12	11	10	11	3,810.2	3,921.2	4,042.3	>6,921.1	>7,231.1	>6,945.7
pseudo-primesDimacs	11	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-radar	12	2	2	2	0	0	0	>4,800.1	>6,130.2	>4,714.9	>10,800.0	>10,800.0	>10,800.0
pseudo-routing	15	4	4	4	4	4	5	>7,429.7	>8,812.7	>7,231.7	>7,247.6	>8,700.4	5,153.4
pseudo-ssa	8	1	1	1	1	1	1	381.6	383.0	381.2	388.5	393.2	387.4
pseudo-ttp	8	2	2	2	2	2	2	463.2	802.9	608.4	2,754.9	4,342.4	2,549.0
pseudo-uclid	39	3	3	3	3	3	3	2,018.1	2,077.3	2,004.6	2,490.7	2,802.8	2,468.0
ramsey3	8	2	2	2	2	2	2	1,724.3	213.0	1,028.2	5.5	7.0	5.7
ramsey4	8	0	0	0	1	3	1	>10,800.0	>10,800.0	>10,800.0	>7,334.2	1,230.1	>7,338.8
rand-10-20-10	20	20	20	20	20	20	20	157.7	157.5	157.7	158.1	158.0	158.1
rand-3-20-20-fcd	50	16	4	15	3	1	3	>42,218.7	>57,614.0	>46,258.1	>61,237.5	>62,354.7	>60,554.3
rand-3-20-20	50	9	4	9	2	1	3	17,959.5	>26,312.1	19,903.8	>28,796.0	>29,973.1	>28,414.3
rand-3-24-24-fcd	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-24-24	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-28-28-fcd	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-28-28	50	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.4 (continued)

		# Solved						ΣCPU (sec)					
	# Instances	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}	PerTuple$^{\Psi}$-as	PerTuple$^{\Psi}$-ns	PerTuple$^{\Psi}$	PerTuple^{m3}-as	PerTuple^{m3}-ns	PerTuple^{m3}
rand-8-20-5	20	1	0	0	0	0	0	2,754.8	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0
renault	2	2	2	2	2	2	2	88.2	138.9	87.7	976.0	1,262.8	974.4
schurrLemma	10	6	7	7	4	3	3	>5,013.9	2,803.8	2,690.0	>16,120.4	>17,132.8	>16,356.5
small	5	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
socialGolfers	12	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0
ssa	8	6	6	6	6	6	6	43.8	43.9	43.7	46.0	47.1	45.9
travellingSalesman-20	15	1	0	1	7	6	7	>27,673.4	>28,800.0	>27,908.3	>7,129.0	>8,964.7	>6,919.7
travellingSalesman-25	15	0	0	0	3	1	4	>14,400.0	>14,400.0	>14,400.0	>9,285.0	>12,739.7	8,809.5
ukVg	65	20	20	20	7	5	7	10,558.0	13,449.6	10,383.6	>55,291.5	>58,460.1	>55,293.4
varDimacs	9	6	6	6	6	6	6	>3,863.3	>3,991.2	>3,913.1	>3,883.5	>3,955.1	>3,880.6
wordsVg	65	42	42	42	37	33	37	>12,322.0	>13,715.1	>12,360.7	>36,254.6	>44,263.5	>34,586.6

Table C.5: Aggregated instances results for PerFB algorithms for all tested binary benchmarks.

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
BH-4-13	7	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
BH-4-4	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
BH-4-7	20	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QCP-10	15	0	0	0	0	12	12	12	12	>43,200.0	>43,200.0	>43,200.0	>43,200.0	683.4	731.8	443.9	463.8
QCP-15	15	0	0	0	0	1	1	1	1	>3,600.0	>3,600.0	>3,600.0	>3,600.0	1,137.2	1,181.6	877.5	893.8
QCP-20	15	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QCP-25	15	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QWH-10	10	7	7	7	7	10	10	10	10	>19,301.1	>19,732.8	>17,143.8	>17,466.2	151.1	160.3	117.0	121.4
QWH-15	10	0	0	0	0	4	4	4	4	>14,400.0	>14,400.0	>14,400.0	>14,400.0	3,526.3	3,661.6	2,715.4	2,762.8
QWH-20	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QWH-25	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bqwh-15-106	100	26	25	26	25	100	100	100	100	>290,575.3	>291,818.5	>288,637.5	>289,816.0	16,387.9	17,762.0	10,083.5	10,597.9
bqwh-18-141	100	0	0	0	0	60	58	64	63	>248,400.0	>248,400.0	>248,400.0	>248,400.0	>119,518.6	>126,466.1	>80,993.5	>84,188.5
coloring	22	18	18	18	18	21	21	18	18	>10,959.0	>10,963.4	>10,951.6	>10,955.9	1,247.2	1,334.2	>10,900.1	>10,902.8
composed-25-1-2	10	10	10	10	10	10	10	10	10	2.0	2.0	2.0	2.0	3.0	3.0	3.0	3.1
composed-25-1-25	10	10	10	10	10	10	10	10	10	2.8	2.9	2.8	2.9	4.4	4.4	4.4	4.4

Table C.5 (continued)

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
composed-25-1-40	10	10	10	10	10	10	10	10	10	3.5	3.5	3.5	3.5	5.0	5.1	5.0	5.0
composed-25-1-80	10	10	10	10	10	10	10	10	10	14.3	15.1	14.4	15.2	14.3	14.9	15.0	15.5
composed-25-10-20	10	0	0	0	0	5	5	5	5	>18,000.0	>18,000.0	>18,000.0	>18,000.0	216.4	229.2	162.7	169.0
composed-75-1-2	10	10	10	10	10	10	10	10	10	7.6	7.6	7.6	7.6	16.8	17.1	18.9	19.2
composed-75-1-25	10	10	10	10	10	10	10	10	10	8.1	8.1	8.1	8.1	19.1	19.5	21.6	22.1
composed-75-1-40	10	10	10	10	10	10	10	10	10	8.9	8.9	8.9	8.9	31.7	32.9	38.5	39.6
composed-75-1-80	10	10	10	10	10	10	10	10	10	22.0	23.1	22.0	23.1	37.2	38.6	44.4	45.8
domino	24	7	7	7	7	7	7	7	7	6,023.6	6,023.5	6,023.7	6,023.7	6,023.7	6,023.8	6,023.9	6,023.7
driver	7	1	1	1	1	4	4	1	1	>10,800.2	>10,800.2	>10,800.1	>10,800.1	4,491.1	4,802.1	>10,800.3	>10,800.3
ehi-85	100	7	5	7	6	50	50	50	50	>176,954.6	>177,921.2	>174,841.5	>175,932.8	>12,593.7	>12,681.7	>12,884.9	>12,960.8
ehi-90	100	7	7	8	8	54	54	54	54	>198,527.5	>199,811.3	>195,193.9	>196,503.5	>16,813.9	>16,942.4	>17,053.5	>17,153.7
fapp-fapp01	11	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
frb30-15	10	0	0	0	0	10	10	10	10	>36,000.0	>36,000.0	>36,000.0	>36,000.0	10,298.8	11,061.6	6,170.4	6,530.5
frb35-17	10	0	0	0	0	0	0	1	2	>7,200.0	>7,200.0	>7,200.0	>7,200.0	>7,200.0	>7,200.0	>6,423.6	5,961.2
frb40-19	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
frb45-21	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.5 (continued)

	# Instances	# Solved								ΣCPU (sec)							
		PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
frb50-23	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
frb53-24	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
frb56-25	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
frb59-26	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
geom	100	24	24	27	27	72	72	81	78	>253,623.3	>255,892.1	>240,073.4	>242,092.9	>51,795.1	>52,830.3	35,060.0	>37,964.8
graphColoring-hos	14	2	2	2	2	6	6	4	4	>14,631.0	>14,605.7	>14,592.2	>14,605.8	1,409.5	1,506.6	>7,305.2	>7,306.4
graphColoring-insertion-full-insertion	41	12	12	12	12	17	17	15	14	>31,428.4	>31,643.0	>31,108.0	>31,291.4	>17,628.8	>18,058.6	>23,171.0	>23,320.8
graphColoring-insertion-k-insertion	32	10	10	10	10	14	14	15	15	>18,727.0	>18,756.1	>18,686.6	>18,707.4	>4,923.6	>4,962.8	3,979.6	4,062.0
graphColoring-leighton-leighton-15	28	5	5	5	5	2	2	0	0	>9,603.6	>9,696.9	>9,604.0	>9,697.7	>21,206.9	>21,437.9	>25,200.0	>25,200.0
graphColoring-leighton-leighton-25	32	4	4	4	4	2	2	0	0	4,504.6	4,706.6	4,505.0	4,707.0	>9,225.0	>9,367.2	>14,400.0	>14,400.0
graphColoring-leighton-leighton-5	8	4	4	4	4	7	7	0	0	>11,130.8	>11,131.3	>11,131.0	>11,131.3	4,771.5	5,091.4	>25,200.0	>25,200.0
graphColoring-mug	8	8	8	8	8	4	4	4	4	366.8	373.8	360.5	365.7	>14,400.5	>14,400.5	>14,400.4	>14,400.4
graphColoring-myciel	16	7	7	7	7	10	9	9	10	>11,969.7	>12,039.4	>11,962.5	>12,032.0	3,973.0	>4,165.1	>3,977.0	2,861.5
graphColoring-register-fpsol	37	3	3	3	3	2	2	0	0	255.3	255.4	255.3	255.6	>5,376.5	>5,510.6	>10,800.0	>10,800.0
graphColoring-register-inithx	32	2	2	2	2	2	2	0	0	>10,959.8	>10,959.9	>10,960.7	>10,960.3	>13,674.7	>13,886.5	>18,000.0	>18,000.0
graphColoring-register-mulsol	49	5	5	5	5	5	5	0	0	>14,489.1	>14,489.2	>14,489.0	>14,489.2	>18,052.9	>18,339.3	>32,400.0	>32,400.0

Table C.5 (continued)

	# Instances	# Solved								ΣCPU (sec)							
		PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
graphColoring-register-zeroin	31	5	5	5	5	3	3	0	0	4,790.2	5,012.2	4,790.2	5,012.2	>8,570.1	>8,670.3	>18,000.0	>18,000.0
graphColoring-school	8	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
graphColoring-sgb-book	26	23	23	23	23	10	10	10	10	>7,566.9	>7,794.2	>7,559.6	>7,785.9	>50,683.2	>50,699.7	>50,592.6	>50,601.1
graphColoring-sgb-games	4	4	4	4	4	2	2	2	2	92.1	100.3	69.5	75.3	>10,010.3	>10,243.6	>8,884.3	>8,976.0
graphColoring-sgb-miles	42	7	7	7	7	7	7	6	6	>5,238.6	>5,318.0	>5,238.2	>5,317.6	>5,768.8	>5,889.7	>8,081.5	>8,119.5
graphColoring-sgb-queen	50	4	4	4	4	5	5	7	6	>11,061.3	>11,080.7	>11,028.1	>11,044.4	>8,462.6	>8,539.2	5,940.1	>6,860.5
hanoi	5	5	5	5	5	5	5	5	5	2,461.4	2,461.4	2,461.6	2,461.3	2,461.2	2,461.2	2,461.5	2,461.3
haystacks	51	8	8	8	8	3	3	3	3	3,062.0	3,214.4	2,970.9	3,118.8	>18,062.6	>18,067.7	>18,052.2	>18,055.5
jobShop-e0ddr1	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-e0ddr2	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-enddr1	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-enddr2	6	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
jobShop-ewddr2	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
knights	19	6	6	6	6	5	4	5	5	>3,793.2	>3,793.7	>3,793.2	>3,793.7	>9,020.8	>11,003.8	>9,022.6	>9,024.0
langford	4	3	3	4	3	2	2	2	2	>5,733.5	>5,809.1	4,207.0	>5,808.2	>7,354.6	>7,362.7	>7,292.1	>7,295.9
langford2	24	10	10	10	10	14	14	15	15	>20,093.2	>20,192.1	>19,987.5	>20,080.8	>6,128.9	>6,216.0	4,089.4	4,273.7

Table C.5 (continued)

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
langford3	23	10	10	10	9	9	9	9	9	2,474.8	2,567.7	2,467.5	>4,015.1	>4,515.1	>4,564.7	>4,125.8	>4,148.5
langford4	24	10	10	9	10	9	9	9	9	1,849.8	1,899.2	>4,008.9	1,899.2	>8,406.7	>8,594.3	>5,982.6	>6,062.2
lard	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
marc	10	5	5	5	5	10	10	5	5	>18,523.3	>18,526.3	>18,523.4	>18,526.3	4,100.2	4,130.8	>18,341.0	>18,341.2
os-taillard-4	30	1	1	1	1	11	11	18	18	>73,614.6	>73,656.8	>73,614.6	>73,656.8	>44,975.4	>45,005.6	>32,154.9	>32,577.1
os-taillard-5	30	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
os-taillard-7	30	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pigeons	25	9	9	9	9	11	11	11	11	>11,804.0	>11,860.5	>11,804.0	>11,860.5	>5,174.1	>5,279.7	>4,913.4	>4,979.5
queenAttacking	10	2	2	2	2	2	2	2	2	4.8	5.2	4.8	5.2	39.6	42.1	26.9	28.2
queens	14	5	5	5	5	7	7	8	8	>11,427.2	>11,455.9	>11,342.4	>11,367.1	>3,888.7	>3,894.7	2,334.8	2,353.9
queensKnights	18	6	6	6	6	6	6	6	6	>9,553.7	>9,686.4	>9,469.1	>9,598.0	>10,519.4	>10,545.2	>9,680.2	>9,692.5
rand-2-23	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-24	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-25	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-26	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-27	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.5 (continued)

	# Instances	# Solved								ΣCPU (sec)							
		PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
rand-2-30-15-fcd	50	1	1	2	1	46	46	48	47	>171,981.8	>172,125.0	>170,501.7	>170,719.3	>57,648.6	>61,388.4	35,372.9	>39,016.0
rand-2-30-15	50	0	0	0	0	38	36	43	45	>162,000.0	>162,000.0	>162,000.0	>162,000.0	>85,079.6	>89,452.4	>56,074.9	58,437.8
rand-2-40-19-fcd	50	0	0	0	0	1	1	2	2	>7,200.0	>7,200.0	>7,200.0	>7,200.0	>4,464.6	>4,524.4	2,691.7	2,834.7
rand-2-40-19	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-50-23-fcd	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-2-50-23	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rlfapGraphs	14	6	6	7	7	9	9	8	8	>12,524.9	>12,567.7	>11,017.2	>11,182.3	5,450.7	5,541.3	>5,200.8	>5,221.0
rlfapGraphsMod	12	8	8	8	8	5	5	5	5	>3,832.1	>3,840.9	>3,818.9	>3,826.9	>14,721.0	>14,732.2	>14,613.0	>14,618.1
rlfapScens11	12	5	5	5	5	2	2	2	2	243.3	250.9	243.9	251.4	>11,456.4	>11,473.1	>11,382.5	>11,395.1
rlfapScens	11	7	7	7	7	8	8	8	8	>4,286.1	>4,315.4	>4,201.2	>4,225.4	2,211.5	2,240.5	1,556.8	1,574.5
rlfapScensMod	13	9	9	9	9	9	9	8	8	>5,073.8	>5,154.3	>4,519.5	>4,568.3	>6,104.0	>6,159.6	>7,423.8	>7,427.9
subs	9	9	9	9	9	9	9	9	9	22.8	23.2	22.4	22.8	159.4	162.4	140.0	142.2
super-jobShop-super-jobShop-e0ddr1	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-e0ddr2	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-enddr1	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
super-jobShop-super-jobShop-enddr2	6	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.5 (continued)

	# Instances	# Solved								ΣCPU (sec)							
		PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
super-jobShop-super-jobShop-ewddr2	10	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
super-os-super-os-taillard-4	30	2	2	2	2	9	9	11	11	>42,933.7	>43,039.0	>42,933.7	>43,039.0	>30,643.4	>30,682.3	>24,858.3	>24,900.4
super-os-super-os-taillard-5	30	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
super-queens	14	4	4	4	4	3	3	3	3	278.3	292.7	278.3	292.7	>3,788.9	>3,804.6	>3,735.4	>3,744.4
tightness0.1	100	0	0	0	0	6	6	10	9	>36,000.0	>36,000.0	>36,000.0	>36,000.0	>25,405.8	>26,195.4	17,949.5	>19,176.2
tightness0.2	100	0	0	0	0	6	5	8	7	>28,800.0	>28,800.0	>28,800.0	>28,800.0	>19,586.4	>20,294.1	12,650.1	>13,874.0
tightness0.35	100	0	0	0	0	9	9	24	21	>90,000.0	>90,000.0	>90,000.0	>90,000.0	>68,057.1	>68,826.0	>50,235.7	>53,164.6
tightness0.5	100	0	0	0	0	11	11	20	20	>72,000.0	>72,000.0	>72,000.0	>72,000.0	>50,941.3	>52,087.4	34,069.5	35,699.3
tightness0.65	100	7	7	13	12	27	26	53	51	>182,037.7	>182,256.9	>171,769.7	>171,914.3	>135,141.1	>137,080.0	96,619.5	>99,910.1
tightness0.8	100	64	64	71	72	55	55	67	65	>111,777.4	>112,868.0	>89,559.9	>90,746.4	>140,980.0	>142,969.9	>100,109.4	>101,675.3
tightness0.9	100	83	83	84	83	70	69	75	74	>84,043.5	>84,925.5	>71,378.0	>73,425.7	>132,899.1	>134,886.5	>102,873.4	>103,709.5

Table C.6: Aggregated instances results for PerFB algorithms for all tested non-binary benchmarks.

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
QG3	7	7	7	7	7	7	7	7	7	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
QG4	7	7	7	7	7	7	7	7	7	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
QG5	7	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QG6	7	7	7	7	7	7	7	7	7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7
QG7	7	7	7	7	7	7	7	7	7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7
aim-100	24	8	8	8	8	17	17	18	18	>42,925.1	>43,678.0	>40,923.2	>41,416.3	>12,801.9	>13,360.8	10,378.3	10,781.8
aim-200	24	1	1	2	2	3	3	3	3	>7,207.4	>7,207.9	>3,940.4	>3,959.2	17.8	18.4	18.6	19.0
aim-50	24	24	24	24	24	24	24	24	24	140.7	158.7	121.5	135.8	102.6	110.1	82.8	86.9
allIntervalSeries	25	7	7	7	7	8	8	9	9	>7,458.1	>7,474.9	>7,412.1	>7,425.9	>4,722.1	>4,807.9	2,934.9	3,095.1
bddLarge	35	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bddSmall	35	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bmc	24	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
bqwh-15-106_glb	100	98	98	98	98	100	100	100	100	>12,708.9	>13,360.9	>10,264.8	>10,597.0	49.1	50.8	48.8	50.1
bqwh-18-141_glb	100	55	56	58	58	100	100	100	100	>183,924.6	>181,034.0	>176,460.3	>172,453.8	193.1	200.9	185.2	191.0
chessboardColoration	20	8	8	8	8	7	7	7	7	>3,773.5	>3,787.2	>3,753.0	>3,764.2	>10,861.5	>11,089.1	>8,994.7	>9,084.8
dag-half	25	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table C.6 (continued)

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
dag-rand	25	22	22	22	22	22	22	22	22	1,563.3	1,565.5	1,563.7	1,566.0	1,639.3	1,647.6	1,644.4	1,652.3
dubois	13	3	3	3	3	3	3	3	3	4,263.6	4,295.1	4,243.5	4,266.3	4,263.1	4,294.7	4,240.6	4,266.7
golombRulerArity3	14	0	0	0	0	2	2	2	2	>7,200.0	>7,200.0	>7,200.0	>7,200.0	453.6	480.5	272.5	286.4
golombRulerArity4	14	2	2	2	2	2	2	2	2	414.4	414.4	414.4	414.4	415.2	415.3	415.4	415.4
graceful	4	1	1	1	1	1	1	1	1	4.1	4.6	3.6	4.0	13.4	14.5	9.1	9.6
jnhSat	16	0	0	0	0	9	9	2	2	>32,400.0	>32,400.0	>32,400.0	>32,400.0	13,680.2	14,244.9	>27,179.0	>27,256.3
jnhUnsat	34	0	0	0	0	17	16	4	4	>61,200.0	>61,200.0	>61,200.0	>61,200.0	18,223.4	>18,775.1	>47,260.8	>47,274.2
latinSquare	10	4	4	4	4	5	5	5	5	>3,601.8	>3,601.8	>3,601.8	>3,601.8	153.3	130.4	158.8	136.2
lexVg	63	47	46	47	46	42	42	42	42	11,121.6	>11,863.7	10,001.5	>10,663.2	>27,094.5	>27,228.8	>24,203.7	>24,286.0
mknap	6	2	2	2	2	2	2	2	2	122.4	122.4	122.4	122.4	122.4	122.4	122.4	122.4
modifiedRenault	50	50	50	50	50	50	50	50	50	3,371.0	3,412.8	2,678.0	2,705.5	10,468.1	10,589.3	6,303.8	6,340.2
nengfa	10	1	1	1	1	2	2	2	2	>3,724.6	>3,725.6	>3,719.4	>3,719.9	427.0	437.5	285.1	289.8
ogdVg	65	18	19	19	19	17	17	16	16	>9,010.5	9,082.9	6,412.6	6,688.5	>11,713.4	>11,734.9	>13,229.0	>13,241.7
ortholatin	9	1	1	1	1	1	1	1	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pret	8	4	4	4	4	4	4	4	4	7,860.8	7,963.2	7,593.9	7,654.1	7,859.9	7,964.1	7,593.8	7,656.4
primes-10	32	12	12	12	12	12	12	12	12	919.2	919.4	918.9	919.0	910.5	910.5	910.6	910.7

Table C.6 (continued)

	# Instances	# Solved								ΣCPU (sec)							
		PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
primes-15	32	8	8	8	8	8	8	8	8	10.8	10.8	10.8	10.8	10.9	10.9	10.9	10.9
primes-20	32	8	8	8	8	8	8	8	8	82.4	82.5	82.5	82.5	82.9	82.9	82.9	82.9
primes-25	32	8	8	8	8	8	8	8	8	111.9	112.0	112.1	112.1	112.7	112.7	112.4	112.5
primes-30	32	6	6	6	6	6	6	6	6	248.6	248.6	248.7	248.7	248.6	248.6	248.7	248.7
pseudo-aim	48	13	13	13	13	25	25	25	25	>46,049.8	>46,199.1	>45,713.3	>45,805.7	3,605.4	3,750.9	3,068.6	3,136.3
pseudo-chnl	21	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-circuits	7	2	2	2	2	2	2	2	2	>3,600.2	>3,600.2	>3,600.1	>3,600.2	>3,600.3	>3,600.3	>3,600.2	>3,600.2
pseudo-fpga	36	11	11	13	13	13	13	13	13	>12,980.0	>13,402.9	2,370.8	2,459.0	541.9	558.5	479.7	488.6
pseudo-garden	7	6	6	6	6	6	6	6	6	3.3	3.6	2.7	2.9	3.7	3.9	3.5	3.6
pseudo-ii	41	1	1	1	1	1	1	1	1	0.5	0.5	0.5	0.5	0.6	0.6	0.6	0.6
pseudo-jnh	16	0	0	0	0	8	8	6	6	>32,400.0	>32,400.0	>32,400.0	>32,400.0	>14,734.0	>15,286.6	>19,084.7	>19,448.0
pseudo-logic-synthesis	17	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-mps	49	6	6	6	6	6	6	6	6	2,018.7	2,018.8	2,017.6	2,017.7	2,017.7	2,017.8	2,017.6	2,017.6
pseudo-mpsReduced	105	1	1	1	1	1	1	1	1	238.7	238.8	237.6	237.6	237.7	237.7	237.6	237.6
pseudo-niklas	19	2	2	2	2	2	2	2	2	2,765.0	2,778.6	2,508.5	2,514.6	1,818.6	1,819.5	1,813.1	1,813.7
pseudo-par	30	12	12	12	12	10	10	10	10	4,195.2	4,409.3	4,153.5	4,362.4	>7,231.0	>7,232.3	>7,228.4	>7,229.1

Table C.6 (continued)

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
pseudo-primesDimacs	11	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pseudo-radar	12	2	2	2	2	3	3	3	3	>4,907.8	>4,954.3	>4,109.4	>4,122.8	546.9	548.5	545.2	546.4
pseudo-routing	15	4	4	4	4	5	5	5	5	>7,399.3	>7,692.5	>6,363.8	>6,563.9	2,782.3	2,897.0	2,312.6	2,382.8
pseudo-ssa	8	1	1	1	1	1	1	1	1	386.8	387.5	384.3	385.1	398.3	399.9	391.7	392.3
pseudo-ttp	8	2	2	2	2	2	2	2	2	886.5	1,012.1	654.4	741.4	4,591.8	4,971.5	2,996.9	3,150.7
pseudo-uclid	39	3	3	3	3	3	3	3	3	2,045.1	2,046.4	1,974.8	1,975.4	2,218.2	2,220.1	2,027.2	2,028.2
ramsey3	8	2	2	1	1	2	2	2	2	812.1	876.3	>3,604.7	>3,605.2	5.7	6.0	5.3	5.6
ramsey4	8	0	0	0	0	3	3	1	1	>10,800.0	>10,800.0	>10,800.0	>10,800.0	862.6	897.6	>7,314.2	>7,317.8
rand-10-20-10	20	20	20	20	20	20	20	20	20	161.6	161.6	161.7	161.7	162.7	162.7	162.8	162.9
rand-3-20-20-fcd	50	5	4	16	16	4	4	6	6	>57,338.8	>57,613.7	>44,713.1	>46,036.0	>56,725.4	>56,841.9	>51,859.7	>52,003.2
rand-3-20-20	50	4	4	9	9	3	3	3	3	>25,933.8	>26,223.6	19,041.7	19,777.3	>26,572.4	>26,656.3	>24,287.1	>24,325.1
rand-3-24-24-fcd	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-24-24	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-28-28-fcd	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-3-28-28	50	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rand-8-20-5	20	0	0	0	0	0	0	0	0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0

Table C.6 (continued)

		# Solved								ΣCPU (sec)							
	# Instances	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os	PerFB$^{\Psi}$-ns-r	PerFB$^{\Psi}$-ns	PerFB$^{\Psi}$-os-r	PerFB$^{\Psi}$-os	PerFB^{m3}-ns-r	PerFB^{m3}-ns	PerFB^{m3}-os-r	PerFB^{m3}-os
renault	2	2	2	2	2	2	2	2	2	142.1	143.0	91.8	92.3	654.1	655.3	460.0	460.6
schurrLemma	10	7	6	7	6	3	3	4	4	2,702.5	>5,105.4	2,607.0	>4,998.8	>16,303.3	>16,435.7	>15,384.5	>15,642.5
small	5	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
socialGolfers	12	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ssa	8	6	6	6	6	6	6	6	6	43.2	43.4	42.9	43.0	46.7	47.3	45.5	45.7
travellingSalesman-20	15	2	1	2	2	7	7	8	7	>27,879.3	>28,227.4	>27,310.7	>27,718.0	>7,549.5	>7,650.6	6,136.4	>6,765.5
travellingSalesman-25	15	0	0	0	0	3	3	3	3	>14,400.0	>14,400.0	>14,400.0	>14,400.0	>10,618.8	>10,843.4	>8,552.7	>8,628.0
ukVg	65	20	20	20	20	13	13	15	15	11,521.7	12,421.5	9,377.5	10,212.5	>31,411.5	>31,479.8	>28,878.2	>28,984.3
varDimacs	9	6	6	6	6	6	6	6	6	>4,002.8	>4,054.9	>3,897.5	>3,934.3	>4,006.5	>4,032.8	>3,925.1	>3,938.9
wordsVg	65	41	41	42	41	39	39	40	40	>13,747.1	>14,183.4	>12,501.6	>13,060.0	>22,704.9	>22,788.2	>20,192.6	>20,268.9
#VALUE!	0	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Summar	0	607	605	630	627	707	706	691	690	>698,333.5	>703,928.2	>644,046.7	>648,962.0	>352,195.7	>356,729.1	>373,541.3	>376,535.1

Appendix D

Per-Benchmark Results for Dangle Identification Algorithms

Table D.1 shows detailed results for the dangle identification algorithms tested in Chapter 6.

Table D.1: Aggregated instance results for dangle identification algorithms for all tested benchmarks using $\frac{|\texttt{dom}|}{\texttt{wdeg}}$.

		# Solved										ΣCPU (sec)									
	# Instances	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r
QG3	7	7	7	7	7	7	7	7	7	7	7	0.6	0.6	0.4	0.7	0.7	0.6	0.6	0.4	0.7	0.7
QG4	7	7	7	7	7	7	7	7	7	7	7	0.6	0.6	0.4	0.7	0.7	0.6	0.6	0.4	0.7	0.7
QG5	7	0	0	0	0	0	0	0	0	0	0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
QG6	7	7	7	7	7	7	7	7	7	7	7	0.8	0.8	0.6	0.9	0.9	0.8	0.8	0.6	0.9	0.9
QG7	7	7	7	7	7	7	7	7	7	7	7	0.8	0.8	0.6	0.9	0.9	0.8	0.8	0.6	0.9	0.9
aim-100	24	10	24	24	15	24	6	16	24	8	16	>53,318.2	5,769.6	4.9	>36,890.8	5,852.3	>65,795.8	>30,456.8	8.2	>61,651.3	>30,375.3
aim-200	24	5	13	24	7	13	5	10	23	6	10	>68,411.4	>43,595.1	260.8	>61,274.9	>43,324.3	>68,411.5	>53,020.9	>5,373.2	>64,819.9	>52,827.5
aim-50	24	24	24	24	24	24	24	24	24	24	24	343.9	30.3	1.2	38.7	32.5	422.0	51.8	1.0	62.2	59.4
allIntervalSeries	25	6	14	17	8	14	6	14	17	8	14	>40,828.7	>11,866.3	2,839.7	>34,106.8	>11,239.5	>40,839.6	>13,163.7	1,042.6	>34,115.4	>12,004.8
bddLarge	35	0	0	0	0	0	0	0	0	0	0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>126,000.0
bddSmall	35	0	0	35	0	0	0	0	35	0	0	>126,000.0	>126,000.0	1,604.7	>126,000.0	>126,000.0	>126,000.0	>126,000.0	1,593.9	>126,000.0	>126,000.0
bmc	24	0	0	1	0	0	0	0	1	0	0	>3,600.0	>3,600.0	82.4	>3,600.0	>3,600.0	>3,600.0	>3,600.0	81.5	>3,600.0	>3,600.0
bqwh-15-106_glb	100	46	100	100	99	100	45	100	100	99	100	>219,648.9	55.6	3.4	>4,811.9	46.2	>220,447.2	56.1	3.3	>4,807.8	46.4
bqwh-18-141_glb	100	6	100	100	60	100	6	100	100	61	100	>338,931.1	258.3	5.7	>169,449.4	168.3	>338,927.7	257.8	5.5	>171,195.6	168.2

Table D.1 (continued)

	# Instances	# Solved										ΣCPU (sec)									
		Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r
chessboardColoration	20	6	8	11	8	8	6	7	12	8	7	>25,366.2	>22,419.4	>8,436.6	>18,162.7	>20,538.1	>25,366.2	>24,872.1	>4,445.7	>18,161.6	>23,698.2
dag-half	25	0	0	16	0	0	0	0	21	0	0	>82,800.0	>82,800.0	>46,356.7	>82,800.0	>82,800.0	>82,800.0	>82,800.0	>14,309.0	>82,800.0	>82,800.0
dag-rand	25	22	22	25	22	22	22	22	25	22	22	>12,567.2	>12,567.7	733.0	>12,564.6	>12,564.9	>12,567.2	>12,567.7	733.0	>12,564.6	>12,564.9
dubois	13	5	5	9	5	5	4	4	6	4	4	>19,217.6	>19,218.4	6,453.9	>19,182.4	>19,182.4	>22,798.2	>22,798.6	>15,412.2	>22,796.7	>22,797.1
golombRulerArity3	14	0	2	8	0	3	0	2	9	0	2	>36,000.0	>29,243.2	>7,957.8	>36,000.0	>26,637.5	>36,000.0	>29,452.5	>4,369.2	>36,000.0	>29,062.2
golombRulerArity4	14	2	2	2	2	2	2	2	2	2	2	414.3	414.5	413.8	414.4	414.6	414.3	414.5	413.8	414.4	414.6
graceful	4	1	1	3	1	2	1	1	3	1	1	>7,273.7	>7,210.1	1,138.4	>7,205.5	>5,245.5	>7,294.5	>7,217.8	418.8	>7,207.6	>7,213.7
jnhSat	16	0	3	16	0	2	0	3	16	0	3	>57,600.0	>52,844.2	44.2	>57,600.0	>52,981.5	>57,600.0	>54,891.8	15.7	>57,600.0	>50,406.3
jnhUnsat	34	2	11	34	2	4	2	11	34	2	4	>115,218.6	>96,971.7	77.2	>115,218.5	>108,296.8	>115,218.6	>98,199.3	33.0	>115,218.5	>108,296.7
latinSquare	10	4	5	5	4	5	4	5	5	4	5	>3,607.8	2,434.2	0.7	>3,601.8	145.1	>3,607.8	2,435.1	0.7	>3,601.8	145.6
lexVg	63	36	39	63	47	44	36	39	63	47	44	>105,230.8	>97,023.6	449.3	>68,967.9	>81,986.9	>105,361.8	>98,593.6	405.0	>70,415.4	>82,750.3
mknap	6	2	2	2	2	2	2	2	2	2	2	122.4	122.4	122.4	122.4	122.4	122.4	122.4	122.4	122.4	122.4
modifiedRenault	50	50	50	50	50	50	50	49	50	50	49	544.4	553.7	67.9	965.8	2,812.2	4,111.7	>4,769.9	65.4	4,787.7	>6,986.9
nengfa	10	1	2	4	1	2	1	2	4	1	2	>10,926.6	>7,567.5	394.6	>10,913.9	>7,448.1	>10,926.7	>7,573.9	178.9	>10,914.0	>7,449.4

Table D.1 (continued)

		# Solved										ΣCPU (sec)									
	# Instances	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r
ogdVg	65	8	14	46	19	16	8	14	46	19	16	>140,578.5	>125,363.2	>9,306.0	>106,283.6	>113,463.9	>140,584.2	>125,305.8	>13,055.0	>106,271.3	>113,461.5
ortholatin	9	1	1	1	1	1	1	1	1	1	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
pret	8	4	4	4	4	4	4	4	4	4	4	2,875.3	2,875.3	217.7	2,846.6	2,847.5	4,713.4	4,713.6	303.6	4,827.6	4,827.3
primes-10	32	12	12	12	12	12	12	12	12	12	12	33.7	33.2	8.0	51.7	51.8	33.7	33.2	8.0	51.7	51.7
primes-15	32	8	8	8	8	8	8	8	8	8	8	5.4	5.5	1.1	6.3	6.3	5.4	5.5	1.1	6.3	6.3
primes-20	32	8	8	8	8	8	8	8	8	8	8	14.1	14.2	3.4	31.7	31.7	14.1	14.2	3.3	31.7	31.7
primes-25	32	8	8	8	8	8	8	8	8	8	8	22.7	22.8	4.9	33.8	33.9	22.7	22.7	4.9	33.8	33.9
primes-30	32	6	6	6	6	6	6	6	6	6	6	8.7	8.7	2.7	211.5	211.5	8.7	8.7	2.7	211.5	211.5
pseudo-aim	48	13	32	46	19	32	14	32	43	19	31	>127,860.8	>63,612.9	>10,468.2	>110,004.8	>63,352.9	>124,575.4	>61,029.7	>18,650.8	>106,577.8	>64,358.2
pseudo-chnl	21	0	0	0	0	0	0	0	1	0	0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	>3,600.0	2,761.1	>3,600.0	>3,600.0
pseudo-circuits	7	3	2	3	2	2	3	2	3	2	2	200.3	>3,600.2	21.2	>3,600.1	>3,600.2	201.8	>3,600.2	20.7	>3,600.1	>3,600.2
pseudo-fpga	36	0	0	1	0	0	0	0	2	0	0	>46,800.0	>46,800.0	>45,646.3	>46,800.0	>46,800.0	>46,800.0	>46,800.0	>41,838.9	>46,800.0	>46,800.0
pseudo-garden	7	6	6	6	6	6	6	6	6	6	6	4.4	3.2	0.2	2.6	3.3	4.6	3.4	0.2	3.0	3.6
pseudo-ii	41	6	6	12	6	6	8	8	9	8	8	>28,380.3	>27,040.9	2,407.0	>27,208.8	>27,000.6	>18,120.9	>16,999.0	>10,904.4	>16,878.8	>16,983.7

Table D.1 (continued)

		# Solved										ΣCPU (sec)									
	# Instances	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r
pseudo-jnh	16	0	6	16	0	5	0	6	16	0	5	>57,600.0	>50,155.8	44.4	>57,600.0	>45,798.6	>57,600.0	>49,799.5	17.7	>57,600.0	>45,145.1
pseudo-logic-synthesis	17	0	0	1	0	0	0	0	1	0	0	>3,600.0	>3,600.0	43.8	>3,600.0	>3,600.0	>3,600.0	>3,600.0	43.5	>3,600.0	>3,600.0
pseudo-mps	49	6	6	7	6	6	6	6	7	6	6	>5,604.2	>5,611.0	2,118.0	>5,619.5	>5,619.4	>5,606.6	>5,614.7	2,107.1	>5,622.0	>5,622.5
pseudo-mpsReduced	105	1	1	1	1	1	1	1	1	1	1	239.6	246.3	234.7	239.4	239.2	242.0	250.0	234.7	241.8	242.4
pseudo-niklas	19	2	2	3	2	2	2	2	3	2	2	>5,335.6	>5,360.4	1,816.4	>6,065.1	>5,433.7	>5,336.9	>5,361.6	1,805.0	>6,066.7	>5,439.0
pseudo-par	30	13	11	20	12	11	13	11	20	13	11	>31,264.4	>34,073.5	1,366.8	>32,127.6	>34,243.9	>31,044.7	>34,101.5	856.0	>31,547.6	>34,279.4
pseudo-primesDimacs	11	0	0	1	0	0	0	0	2	0	0	>14,400.0	>14,400.0	>10,818.5	>14,400.0	>14,400.0	>14,400.0	>14,400.0	>8,519.4	>14,400.0	>14,400.0
pseudo-radar	12	3	3	6	3	3	3	3	6	3	3	>11,589.7	>11,426.2	158.2	>13,491.1	>11,253.6	>11,649.4	>11,427.5	156.9	>13,496.1	>11,255.1
pseudo-routing	15	3	3	6	4	5	3	3	6	3	5	>18,243.9	>18,397.7	>4,081.7	>16,907.7	>11,526.4	>18,862.5	>18,780.1	>3,610.5	>17,564.0	>12,944.0
pseudo-ssa	8	2	2	7	2	2	2	2	7	2	2	>18,152.2	>18,155.0	316.3	>18,151.8	>18,155.4	>18,319.1	>18,326.6	314.4	>18,319.8	>18,327.9
pseudo-ttp	8	2	2	2	2	2	2	2	2	2	2	868.6	423.1	3.0	339.7	728.7	782.4	514.8	1.2	298.6	2,200.7
pseudo-uclid	39	3	3	8	3	3	3	3	8	3	3	>19,826.5	>19,826.6	2,581.9	>19,967.7	>20,020.1	>19,837.4	>19,837.6	3,346.4	>19,978.6	>20,031.0
ramsey3	8	0	2	2	1	2	0	2	2	1	2	>7,200.0	6.7	0.2	>3,604.4	6.2	>7,200.0	9.8	0.1	>3,606.5	9.2
ramsey4	8	0	1	2	0	1	0	1	1	0	1	>14,400.0	>10,968.7	>9,045.2	>14,400.0	>10,917.9	>14,400.0	>10,996.2	>10,802.6	>14,400.0	>10,934.5

Table D.1 (continued)

		# Solved										ΣCPU (sec)									
	# Instances	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r
rand-10-20-10	20	20	20	20	20	20	20	20	20	20	20	45.1	45.1	7.6	44.9	44.9	45.1	45.1	7.6	44.9	44.9
rand-3-20-20-fcd	50	0	4	50	16	7	0	4	50	15	5	>180,000.0	>173,092.0	897.3	>161,694.3	>168,838.4	>180,000.0	>173,569.3	851.0	>161,546.7	>169,894.8
rand-3-20-20	50	0	1	50	6	3	0	0	50	6	1	>180,000.0	>179,764.8	1,763.3	>169,810.8	>177,595.3	>180,000.0	>180,000.0	1,546.2	>169,738.3	>179,066.8
rand-3-24-24-fcd	50	0	0	41	0	0	0	0	40	0	0	>147,600.0	>147,600.0	29,742.2	>147,600.0	>147,600.0	>147,600.0	>147,600.0	>30,171.8	>147,600.0	>147,600.0
rand-3-24-24	50	0	0	30	0	0	0	0	28	0	0	>126,000.0	>126,000.0	>53,995.2	>126,000.0	>126,000.0	>126,000.0	>126,000.0	>46,527.5	>126,000.0	>126,000.0
rand-3-28-28-fcd	50	0	0	7	0	0	0	0	7	0	0	>32,400.0	>32,400.0	>15,939.1	>32,400.0	>32,400.0	>32,400.0	>32,400.0	>17,119.0	>32,400.0	>32,400.0
rand-3-28-28	50	0	0	4	0	0	0	0	4	0	0	>14,400.0	>14,400.0	6,002.6	>14,400.0	>14,400.0	>14,400.0	>14,400.0	4,118.1	>14,400.0	>14,400.0
rand-8-20-5	20	0	0	20	0	0	0	0	20	1	0	>72,000.0	>72,000.0	1,487.2	>72,000.0	>72,000.0	>72,000.0	>72,000.0	420.7	>71,028.2	>72,000.0
renault	2	2	2	2	2	2	2	2	2	2	2	15.0	19.1	2.8	92.4	460.6	15.1	19.2	2.7	92.5	460.7
schurrLemma	10	6	5	9	7	5	6	4	9	7	4	>14,348.1	>20,290.8	1,943.9	>9,807.6	>20,042.2	>14,348.1	>22,706.0	609.9	>9,807.6	>23,139.9
small	5	0	0	3	0	0	0	0	3	0	0	>14,400.0	>14,400.0	>3,778.4	>14,400.0	>14,400.0	>14,400.0	>14,400.0	>3,755.1	>14,400.0	>14,400.0
socialGolfers	12	0	0	2	0	0	0	0	2	0	0	>10,800.0	>10,800.0	>3,701.1	>10,800.0	>10,800.0	>10,800.0	>10,800.0	>4,111.9	>10,800.0	>10,800.0
ssa	8	4	4	7	4	4	6	6	7	6	6	>11,176.7	>11,187.2	906.5	>11,174.7	>11,187.2	>3,821.4	>3,826.1	228.1	>3,821.0	>3,826.6
travellingSalesman-20	15	0	9	15	2	9	0	8	15	1	8	>54,000.0	>29,426.4	125.0	>48,312.3	>27,877.2	>54,000.0	>30,227.3	85.4	>50,651.4	>29,143.2

Table D.1 (continued)

		# Solved										ΣCPU (sec)									
	# Instances	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r	Di-AllSolFB$^{\Psi}$	Di-AllSolFB^{m3}	Di-CT	Di-PerFB$^{\Psi}$-os-r	Di-PerFB^{m3}-os-r	H-AllSolFB$^{\Psi}$	H-AllSolFB^{m3}	H-CT	H-PerFB$^{\Psi}$-os-r	H-PerFB^{m3}-os-r
travellingSalesman-25	15	0	2	15	0	6	0	1	15	0	4	>54,000.0	>49,275.6	2,110.3	>54,000.0	>43,831.0	>54,000.0	>50,753.2	995.3	>54,000.0	>47,960.5
ukVg	65	7	11	40	21	16	7	11	40	21	15	>129,564.2	>116,415.2	>13,121.2	>81,746.6	>99,112.5	>129,608.3	>116,881.3	>10,143.2	>82,020.8	>101,585.7
varDimacs	9	7	7	8	7	7	6	6	8	6	6	>8,920.8	>8,600.6	>3,915.8	>9,061.9	>8,647.9	>11,034.3	>11,017.8	>3,982.2	>11,075.1	>11,040.5
wordsVg	65	30	37	65	42	41	30	37	65	42	41	>133,815.8	>110,286.7	1,407.6	>90,597.8	>98,167.3	>133,818.2	>110,427.5	1,347.3	>90,670.2	>98,226.8

Bibliography

Daniel Aarno. Templatized C++ Command Line Parser, 2022. Accessed: 2022-09-11.

Amine Balafrej, Christian Bessière, Gilles Trombettoni, and El Houssine Bouyakhf. Adaptive Singleton-based Consistencies. In *AAAI Conference on Artificial Intelligence*, pages 2601–2607, July 2014.

Amine Balafrej, Christian Bessière, and Anastasia Paparrizou. Multi-Armed Bandits for Adaptive Constraint Propagation. In *Proc. of IJCAI 2015*, pages 290–296, 2015.

Ken Bayer, Josh Snyder, and Berthe Y. Choueiry. An Interactive Constraint-Based Approach to Minesweeper. In *Proceedings of AAAI-2006*, pages 1933–1934, 2006.

Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, July 1983.

Hachemi Bennaceur and Mohamed-Salah Affane. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Principles and Practice of Constraint Programming*, pages 560–564. Springer Berlin Heidelberg, 2001.

Christian Bessière, Kostas Stergiou, and Toby Walsh. Domain Filtering Consistencies for Non-Binary Constraints. *Artificial Intelligence*, 172:800–822, 2008.

Christian Bessiere. *Handbook of Constraint Programming*, chapter Constraint Propagation, pages 29–83. Elsevier, 2006.

Bozhena Bidyuk and Rina Dechter. On Finding Minimal w-Cutset Problem. In *Proceedings of the Conference on Uncertainty in AI (UAI 04)*, 2004.

Christian Bliek and Djamilla Sam-Haroud. Path Consistency for Triangulated Constraint Graphs. In *Proceedings of the 16 th International Joint Conference on Artificial Intelligence*, pages 456–461, Stockholm, Sweden, 1999.

Boost. Boost C++ Libraries. http://www.boost.org/, 2022. Accessed: 2022-09-30.

Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proc. ECAI 2004*, pages 146–150, 2004.

Preston Briggs and Linda Torczon. An Efficient Representation for Sparse Sets. *ACM Lett. Program. Lang. Syst.*, 2(1-4):59–69, March 1993.

James O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7(2):24–27, feb 1995.

Romuald Debruyne and Christian Bessière. From Restricted Path Consistency to Max-Restricted Path Consistency. In *Principles and Practice of Constraint Programming (CP 97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 312–326. Springer, 1997.

Romuald Debruyne and Christian Bessière. Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In *Proceedings of the 15 th International Joint Conference on Artificial Intelligence*, pages 412–417, 1997.

Romuald Debruyne and Christian Bessière. Domain Filtering Consistencies. *Journal of Artificial Intelligence Research*, 14:205–230, 2001.

Romuald Debruyne. A Strong Local Consistency for Constraint Satisfaction. In *Proceedings of ICTAI 99*, pages 202–209, 1999.

Rina Dechter and Judea Pearl. The Cycle-Cutset Method for improving Search Performance in AI Applications. In *Third IEEE Conference on AI Applications*, pages 224–230, Orlando, FL, 1987.

Rina Dechter and Judea Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, 34:1–38, 1988.

Rina Dechter and Judea Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38:353–366, 1989.

Rina Dechter and Peter van Beek. Local and Global Relational Consistency. *Theor. Comput. Sci.*, 173(1):283–308, 1997.

Rina Dechter, Kalev Kask, and Javier Larrosa. A General Scheme for Multiple Lower Bound Computation in Constraint Optimization. In *Proceedings of the Seventh International Conference on Principle and Practice of Constraint Programming (CP 01)*, pages 346–360, 2001.

Rina Dechter. Bucket Elimination: A Unifying Framework for Probabilistic Inference Algorithms. In *Proceedings of the Conference on Uncertainty in AI (UAI 96)*, pages 211–219, 1996.

Rina Dechter. Mini-Buckets: A General Scheme of Generating Approximations in Automated Reasoning. In *Proceedings of the 15th Joint Conference on Artificial Intelligence (IJCAI 97)*, pages 1297–1302, 1997.

Rina Dechter. *Constraint Processing*, chapter Directional Consistency, pages 91–101. Morgan Kaufmann, 2003.

Rina Dechter. *Constraint Processing.* Morgan Kaufmann, 2003.

Rina Dechter. *Constraint Processing*, chapter Directional Consistency, page 89. Morgan Kaufmann, 2003.

Rina Dechter. *Constraint Processing*, chapter Directional Consistency, page 90. Morgan Kaufmann, 2003.

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 Michel Rueher, editor, *Principles and Practice of Constraint Programming*, pages 207–223, Cham, 2016. Springer International Publishing.

Guilherme Alex Derenievicz and Fabiano Silva. Epiphytic Trees: Relational Consistency Applied to Global Optimization Problems. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 153–169. Springer International Publishing, 2018.

David Duris. Some Characterizations of γ and β-acyclicity of Hypergraphs. *Inf. Process. Lett.*, 112(16):617–620, August 2012.

Eugene C. Freuder and Charles D. Elfe. Neighborhood Inverse Consistency Preprocessing. In *Proceedings of AAAI-96*, pages 202–208, Portland, Oregon, 1996.

Eugene C. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1):24–32, 1982.

Eugene C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29:24–32, 1982.

Eugene C. Freuder. A Sufficient Condition for Backtrack-Bounded Search. *JACM*, 32 (4):755–761, 1985.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

Daniel J. Geschwender, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. A Portfolio Approach for Enforcing Minimality in a Tree Decomposition. In *Proc. of CP 2016*, page 10 pages, 2016.

Daniel J. Geschwender. Effectively Enforcing Minimality During Backtrack Search. Master's thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, May 2018.

Martin C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press Inc., New York, NY, 1980.

Georg Gottlob. On Minimal Constraint Networks. In *Proceedings of 17th International Conference on Principle and Practice of Constraint Programming (CP 11)*, volume 6876 of *Lecture Notes in Computer Science*, pages 325–0339. Springer, 2011.

Marc H. Graham. On the Universal Relation. Technical report, University of Toronto, 1979.

Marc Gyssens. On the Complexity of Join Dependencies. *ACM Trans. Database Systems*, 11(1):81–108, 1986.

Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A Generic Arc Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57:291–321, 1992.

Ian S. Howell, Berthe Y. Choueiry, and Hongfeng Yu. Visualizations to Summarize Search Behavior. In Helmut Simonis, editor, *Proceedings of the 26th International*

Conference on Principles and Practice of Constraint Programming, pages 392–409, 2020.

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 Workshop on Tools for AI*, pages 420–427, 1989.

Philippe Jégou and Cyril Terrioux. Hybrid Backtracking Bounded by Tree-Decomposition of Constraint Networks. *Artificial Intelligence*, 146:43–75, 2003.

Philippe Jégou and Cyril Terrioux. Hybrid Backtracking Bounded by Tree-Decomposition of Constraint Networks. *Artificial Intelligence*, 146:43–75, 2003.

Philippe Jégou, Samba Ndiaye, and Cyril Terrioux. Computing and Exploiting Tree-Decompositions for Solving Constraint Networks. In *CP 05*, pages 777–781, 2005.

Philippe Jégou. On the Consistency of General Constraint-Satisfaction Problems. In *AAAI 1993*, pages 114–119, 1993.

Shant Karakashian, Robert Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *24th AAAI Conference on Artificial Intelligence (AAAI 10)*, pages 101–107, 2010.

Shant Karakashian, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Relational Consistency by Constraint Filtering. In *25th ACM Symposium On Applied Computing (ACM SAC 10)*, pages 2073–2074, Sierre, Switzerland, 2010.

Shant Karakashian, Robert J. Woodward, and Berthe Y. Choueiry. Reformulating R($*, m$)C with Tree Decomposition. In *Ninth International Symposium on Abstrac-*

tion, Reformulation and Approximation (SARA 2011), pages 62–69. AAAI Press, 2011.

Shant Karakashian, Robert J. Woodward, and Berthe Y. Choueiry. Practical Tractability of CSPs by Higher Level Consistency and ree Decomposition. In *Eighteenth International Conference on Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *Lecture Notes in Computer Science*, pages –. Springer, 2012.

Shant Karakashian, Robert Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proceedings of the 27th Conference on Artificial Intelligence (AAAI 2013)*, pages 466–473, 2013.

Shant Karakashian. *Practical Tractability of CSPs by Higher Level Consistency and Tree Decomposition*. PhD thesis, University of Nebraska-Lincoln, 2013.

Kalev Kask, Rina Dechter, Javier Larrosa, and Avi Dechter. Unifying Tree Decompositions for Reasoning in Graphical Models. *Artificial Intelligence*, 166(1-2):165–193, 2005.

Uffe Kjærulff. Triangulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.

Javier Larrosa. Boosting Search with Variable Elimination. In *Proceedings of CP 2000*, volume 1894 of *LNCS*, pages 291–305, 2000.

Vianney le Clément, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *Proc. of the CP Workshop on TRICS 2013*, 2013.

Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. A Path-Optimal GAC Algorithm for Table Constraints. In *Proc. of ECAI 2012*, pages 510–515, 2012.

Christophe Lecoutre, Anastasia Paparrizou, and Kostas Stergiou. Extending STR to a Higher-Order Consistency. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence (AAAI 2013)*, pages 576–582, 2013.

Christophe Lecoutre. STR2: Optimized Simple Tabular Reduction for Table Constraints. *Constraints*, 16(4):341–371, 2011.

Chavalit Likitvivatanavong, Wei Xia, and Roland Yap. Higher-Order Consistencies Through GAC on Factor Variables. In *Proc. of CP 2014*, pages 497–513, 2014.

Chavalit Likitvivatanavong, Wei Xia, and Roland Yap. Decomposition of the Factor Encoding for CSPs. In *Proc. of IJCAI 2015*, pages 353–359, 2015.

Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

David Maier. *The Theory of Relational Databases*. Pitman Publishing Limited, 1983.

Jean-Baptiste Mairy, Yves Deville, and Christophe Lecoutre. Domain k-Wise Consistency Made as Simple as Generalized Arc Consistency. In *Proc. of CPAIOR 2014*, pages 235–250, 2014.

Sylvain Merchezn, Christophe Lecoutre, and Frédéric Boussemart. AbsCon: A Prototype to Solve CSPs with Abstraction. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 730–744, 2001.

Ugo Montanari. Networks of Constraints: Fundamental Properties and Application to Picture Processing. *Information Sciences*, 7:95–132, 1974.

Bernard A. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5:188–224, 1989.

Anastasia Paparrizou and Kostas Stergiou. An Efficient Higher-Order Consistency Algorithm for Table Constraints. In *Proc. AAAI 2012*, 2012.

Anastasia Paparrizou and Kostas Stergiou. Strong local consistency algorithms for table constraints. *Constraints*, 21(2):163–197, Apr 2016.

Anastasia Paparrizou and Kostas Stergiou. On Neighborhood Singleton Consistencies. In *Proc. of IJCAI 2017*, pages 736–742, 2017.

Guilluame Perez and Jean-Charles Régin. Improving GAC-4 for Table and MDD Constraints. In *Proc. of CP 2014*, pages 606–621, 2014.

Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9 (3):268–299, 1993.

Charles Prud'homme. Choco Solver Search Loop Documentation. https://choco-solver.org/docs/advanced-usages/search-loop/, 2022. Accessed: 2022-09-25.

Charles Prud'homme. Choco Solver v4.0.6: N-Ary Constraint Documentation. https://javadoc.io/doc/org.choco-solver/choco-solver/4.0.6/org/chocosolver/solver/constraints/extension/nary/package-summary.html, 2022. Accessed: 2022-09-10.

Irina Rish and Rina Dechter. Resolution versus Search: Two Strategies for SAT. *J. Autom. Reasoning*, 24(1/2):225–275, 2000.

Daniel Sabin and Eugene C. Freuder. Understanding and improving the mac algorithm. In Gert Smolka, editor, *Principles and Practice of Constraint Programming-CP97*, pages 167–181, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

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.

Anthony Schneider and Berthe Y. Choueiry. Pw-ct: Extending compact-table to enforce pairwise consistency on table constraints. In John Hooker, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 345–361. Springer International Publishing, 2018.

Anthony Schneider, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Improving relational consistency algorithms using dynamic relation partitioning. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 688–704, Cham, 2014. Springer International Publishing.

Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and Programming With Gecode v.6.2.0. https://www.gecode.org/doc-latest/MPG.pdf, 2022. Accessed: 2022-09-10, page 71.

Kostas Stergiou. Strong Inverse Consistencies for Non-Binary CSPs. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, volume 1 of *ICTAI 07*, pages 215–222, 2007.

Trieu Hung Tran. Modeling and Solving the Nonogram Puzzle Using Constraint Programming. Undergraduate Thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2019.

Julian R. Ullmann. Partition Search for Non-binary Constraint Satisfaction. *Information Sciences*, 177(18):3639–3678, September 2007.

Julien Vion, Thierry Petit, and Narendra Jussien. Integrating Strong Local Consistencies into Constraint Solvers. In *Recent Advances in Constraints*, pages 90–104, 2011.

Richard J. Wallace. SAC and Neighbourhood SAC. *AI Communications*, 28(2):345–364, January 2015.

David Waltz. Understanding Line Drawings of Scenes with Shadows. In P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, Inc., 1975.

Ruiwei Wang, Wei Xia, Roland H. C. Yap, and Zhanshan Li. Optimizing simple tabular reduction with a bitwise representation. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 787–793. AAAI Press, 2016.

Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Neighborhood Inverse Consistency as Lookahead for Non-Binary CSPs. In *25th AAAI Conference on Artificial Intelligence (AAAI 11)*, pages 1830–1831, 2011.

Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *25th AAAI Conference on Artificial Intelligence (AAAI 11)*, pages 112–119, 2011.

Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Reformulating the Dual Graphs of CSPs to Improve the Performance of Relational

Neighborhood Inverse Consistency. In *Ninth International Symposium on Abstraction, Reformulation and Approximation (SARA 2011)*, pages 140–148. AAAI Press, 2011.

Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Revisiting Neighborhood Inverse Consistency on Binary CSPs. In *Eighteenth International Conference on Principles and Practice of Constraint Programming (CP 2012)*, volume 7514 of *Lecture Notes in Computer Science*, pages 688–703. Springer, 2012.

Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Cycle-Based Singleton Local Consistencies. In *Proc. of AAAI 2017*, pages 5005–5006, 2017.

Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. A Reactive Strategy for High-Level Consistency During Search. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 1390–1397, 2018.

Robert Woodward. *Higher Level Consistencies: Where, When, and How Much.* PhD thesis, University of Nebraska-Lincoln, 2018.

Clement T. Yu and Meral Z. Ozsoyoglu. An Algorithm for Tree-Query Membership of a Distributed Query. In *Proceedings of the Third IEEE International Conference on Computer Software and Applications (COMPSAC 1979)*, pages 306–312, 1979.