Higher-Level Consistencies : When, Where, and How Much

Robert J. Woodward

To cite this version:

Robert J. Woodward. Higher-Level Consistencies : When, Where, and How Much. Data Structures and Algorithms [cs.DS]. Université Montpellier; University of Nebraska-Lincoln, 2018. English. NNT : 2018MONTS145 . tel-02295985

HAL Id: tel-02295985
https://theses.hal.science/tel-02295985

Submitted on 24 Sep 2019

HIGHER-LEVEL CONSISTENCIES: WHERE, WHEN, AND HOW MUCH

by

Robert J. Woodward

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 and
Dr. Christian Bessiere

Lincoln, Nebraska

September, 2018

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITÉ DE MONTPELLIER

En informatique

École doctorale Information, Structures, Systèmes

Unité de recherche Laboratoire d'Informatique,
de Robotique et de Micro-électronique de Montpellier (LIRMM)

En partenariat international avec Université du Nebraska--Lincoln, États Unis

Les Cohérences Fortes : Où, Quand, et Combien

Présentée par Robert J. WOODWARD
Le 20 septembre 2018

Sous la direction de Christian BESSIERE
et Berthe Y. CHOUEIRY

Devant le jury composé de

Sébastian ELBAUM, Professeur, Université de Virginie	rapporteur
Stephen D. SCOTT, Professeur Associé, Université du Nebraska—Lincoln	rapporteur
Souhila KACI, Professeur, LIRMM	examinatrice
Jamie RADCLIFFE, Professeur, Université du Nebraska—Lincoln	examinateur
Christian BESSIERE, directeur de recherche CNRS, LIRMM	co-directeur
Berthe Y. CHOUEIRY, Professeur Associé, Université du Nebraska—Lincoln	co-directrice

UNIVERSITÉ
DE MONTPELLIER

HIGHER-LEVEL CONSISTENCIES: WHERE, WHEN, AND HOW MUCH

Robert J. Woodward, Ph. D.

University of Nebraska, 2018

Adviser: B.Y. Choueiry and C. Bessiere

Determining whether or not a Constraint Satisfaction Problem (CSP) has a solution is $\mathcal{NP}$-complete. CSPs are solved by inference (i.e., enforcing consistency), conditioning (i.e., doing search), or, more commonly, by interleaving the two mechanisms. The most common consistency property enforced during search is Generalized Arc Consistency (GAC). In recent years, new algorithms that enforce consistency properties stronger than GAC have been proposed and shown to be necessary to solve difficult problem instances.

We frame the question of balancing the cost and the pruning effectiveness of consistency algorithms as the question of determining *where*, *when*, and *how much* of a higher-level consistency to enforce during search. To answer the 'where' question, we exploit the topological structure of a problem instance and target high-level consistency where cycle structures appear. To answer the 'when' question, we propose a simple, reactive, and effective strategy that monitors the performance of backtrack search and triggers a higher-level consistency as search thrashes. Lastly, for the question of 'how much,' we monitor the amount of updates caused by propagation and interrupt the process before it reaches a fixpoint. Empirical evaluations on benchmark problems demonstrate the effectiveness of our strategies.

DEDICATION

Dedicated to the memory of Dr. John C. Woodward Sr.

ACKNOWLEDGMENTS

I would like to thank Dr. Berthe Y. Choueiry and Dr. Christian Bessiere for their continued support and encouragement and for allowing me to be a part of both of research groups, namely, the Constraint Systems Laboratory (ConSystLab) at the University of Nebraska-Lincoln and the Coconut team at LIRMM and the Université de Montpellier. I treasure the friendships, conversations, and interactions with all of the people in both labs and feel honored that I could belong to both groups.

I would like to acknowledge research collaborations and the scientific input of the following individuals: Mr. Anthony Schneider for collaboration on Stampede, the ConSystLab solver, without which my research would have not been able to advance so far; Mr. Denis Komissarov laid important foundation for me being able to implement visualizations in Stampede; Mr. Ian Howell extended much of my initial visualization work (Chapter 3) into Wormhole far faster and better looking than I ever could; Mr. Nathan Stender with whom I developed the framework for enforcing multiple consistencies (Section 3.4.2), which was made more efficient in collaboration with Mr. Schneider; Mr. Christopher Reeson created the original △PPC algorithm that I extended (Chapter 6); Mr. Daniel Geschwender was always available for help with experimental design. I am grateful to the Holland Computing Center team for their support in running all of the experiments, especially to Dr. David Swanson and Dr. Derek Weitzel.

Finally, I am grateful to my loving family, who always encouraged me to pursue my passion of Computer Science. I am especially grateful to my wife, Allison, who lovingly and patiently put up with all the late nights spent on research.

GRANT INFORMATION

This research was supported by:

- National Science Foundation (NSF) Grants No. RI-111795 and RI-1619344,
- An NSF Graduate Research Fellowship Grant No. 1041000,
- An NSF Graduate Research Opportunities Worldwide grant,
- A Chateaubriand Fellowship of the Office for Science and Technology, Embassy of France in the USA, and
- The Dean's Fellowship, Office of Graduate Studies, University of Nebraska-Lincoln.

This work was completed utilizing the Holland Computing Center of the University of Nebraska, which receives support from the Nebraska Research Initiative.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Constraint Processing (CP) is a flexible and effective framework for modeling and solving many decision and optimization problems in Engineering, Computer Science, and Management. In contrast to other areas that study the same problems, such as Mathematical Programming and SAT solving, the formulation of a Constraint Satisfaction Problem (CSP) allows the user to state arbitrary constraints over a set of variables in a transparent way, thus, directly reflecting the human's understanding of the problem.

Many combinatorial problems of practical importance are commonly modeled as Constraint Satisfaction Problems (CSPs), including scheduling [Baptiste *et al.*, 2006], resource allocation [Lim *et al.*, 2004], and product configuration and design [Yvars, 2008]. Puzzles are whimsical and attractive tools to introduce the general public to CSPs and also to attract Computer Science students to this area of study. Examples include the Sudoku puzzle [Reeson *et al.*, 2007; Howell *et al.*, 2018a],[1] Minesweeper [Bayer *et al.*, 2006],[2] and the Game of Set [Swearingn *et al.*, 2011].[3]

---

[1] `http://sudoku.unl.edu`

[2] `http://minesweeper.unl.edu`

[3] `http://gameofset.unl.edu`

Research on CP dates back to the early 1960's, and the field has matured into an independent research area in Artificial Intelligence with textbooks [Tsang, 1993; Dechter, 2003a; Lecoutre, 2009], a handbook [Rossi *et al.*, 2006], an association,[4] a journal,[5] and an annual conference.[6]

To solve a CSP, CP focuses on two main directions: search and inference. In this dissertation, we use constructive backtrack search as a sound and complete algorithm for solving CSPs. Inference relies on a set of consistency properties and algorithms for enforcing them. These properties and algorithms are perhaps what best distinguishes CP from related fields that address the same combinatorial problems. They constitute the focus of this dissertation.

1.1 Motivation and Claims

Consistency algorithms operate by removing from the problem values or combination of values that cannot possibly appear in a solution to the problem. They typically operate locally on subproblems of a fixed size. As such, they typically run in polynomial time in the number of variables in the problem. In practice, they are interleaved with search, which runs in exponential time in the number of variables. By pruning the search tree and removing inconsistent branches and subtrees, enforcing consistency can significantly reduce the size of the search space. The stronger the enforced consistency, the larger the pruning (see Figure 1.1). However, the higher the consistency, the higher the computational cost of enforcing it. Thus, it becomes critical to decide whether it is more cost effective to spend more time exploring the search tree or pruning it (see Figure 1.2).

---

[4] Association for Constraint Programming (ACP), `http://www.a4cp.org/`.

[5] Constraints, An International Journal published by Springer.

[6] International Conference on Principles and Practice of Constraint Programming with proceedings published by Springer in their series 'Lecture Notes on Computer Science.'

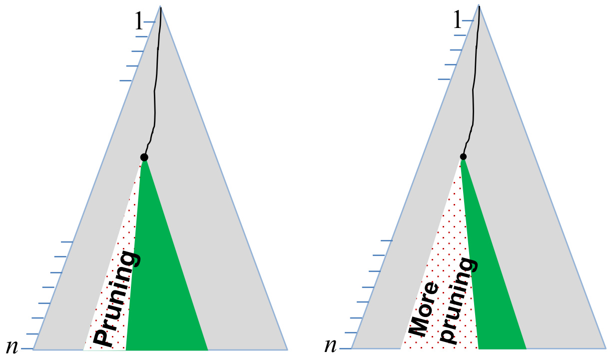


Figure 1.1: The stronger the consistency, the more the pruning

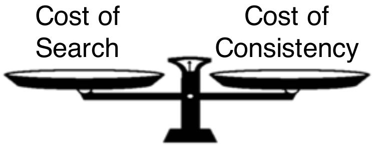


Figure 1.2: Balancing the cost of search and that of consistency

In recent years, effective dynamic variable-ordering heuristics that learn during search have rendered the search cost even more sensitive to that of the algorithms for enforcing higher-level consistency (HLC), especially when these algorithms are applied systematically throughout search and uniformly over the entire network.

In this dissertation, we claim that *strategies for enforcing HLCs during search can be organized along orthogonal dimensions and we have identified three such 'axes,' namely, where, when, and how much of an HLC to enforce,* as shown in Figure 1.3.

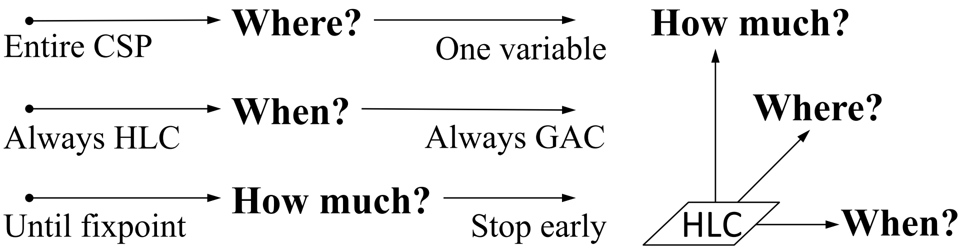


Figure 1.3: Dimensions of enforcing consistency

In summary,

- The 'where' axis identifies specific (or groups of) variables/constraints on which HLC is enforced
- The 'when' axis identifies at what point, during search, HLC is enforced

- The ‘how much’ axis indicates whether or not HLC is forced to terminated before reaching a fixpoint.

The point of origins where these three axes meet indicates the ‘strongest’ application of HLC (i.e., enforce HLC uniformly over the entire future subproblem, at each variable instantiation, and until quiescence). While such a strategy proved useful for solving difficult problem instances, the cost overhead is not always warranted. This situation yields the following question, central to this dissertation:

> *Where, when, and how much of a higher-level consistency should be enforced during search?*

In this dissertation, we answer this critical question as follows:

> High-Level consistency (HLC) properties and algorithms are instrumental for smashing the hardness of a problem instance and are cost effective:
>
> 1. When the search starts thrashing
> 2. Where the local structure of the constraint network has loops
> 3. As long as filtering and propagation are active and ‘alive’

We implement the above vision with a set of techniques that:

1. Monitor the search progress to dynamically enforce higher-level consistency when search appears to be thrashing.
2. Identify critical cycles in the problem's topological structure on which to restrict the application of the higher-level consistency.
3. Monitor the 'liveliness' of the filtering along the propagation queue and terminate propagation early and before a fixpoint is reached.

1.2 Approach

In this dissertation, we propose to combine techniques that 'weaken' HLC along one or more of the three axes identified above (i.e., when, where, and how much) in order to effectively prune the search space while avoiding the cost overhead and maintaining competitive performance. The main components of our techniques are as follows, see Figure 1.4:

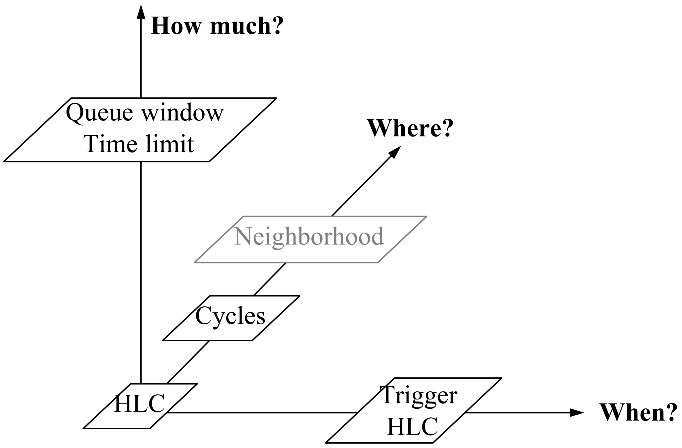


Figure 1.4: The dimensions of enforcing consistency investigated in this dissertation

1. *Monitor search to trigger HLC* (see 'Trigger HLC' in Figure 1.4): We propose a reactive technique that monitors the amount of backtracking steps during

search as an indication of wasteful thrashing. It automatically increases the frequency of applying HLC as long it is effectively pruning the search space. Otherwise, it decreases this frequency.

2. *Identify cycles to channel HLC* (see 'Cycles' in Figure 1.4): We propose to exploit existing cycles in the constraint graph and even create new ones as structures particularly effective at localizing and channeling propagation.

3. *Monitor propagation to interrupt any single execution of HLC* (see 'Queue window' and 'Time limit' in Figure 1.4): We propose to monitor the effectiveness of constraint propagation by watching whether or not any filtering is obtained during a window whose width is a function of the size of the propagation queue. We also bound the maximum duration of any single call to HLC.

Below, we overview each of the proposed techniques.

1.2.1 Visualizing Search and Consistency Costs

In order to illustrate the performance of search in terms of the effort spent searching, thrashing, and enforcing consistency, we propose to visualize:

1. The number of backtracks per depth of the search tree (BpD).

2. The number of calls per depth of the search tree (CpD) to a given consistency algorithm.

Figure 1.5 shows the BpD of a backtrack search with the consistency algorithm APOAC [Balafrej *et al.*, 2014] for problem instance pseudo-aim-200-1-6-4 of the pseudo-aim benchmark.[7]

---

[7] `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

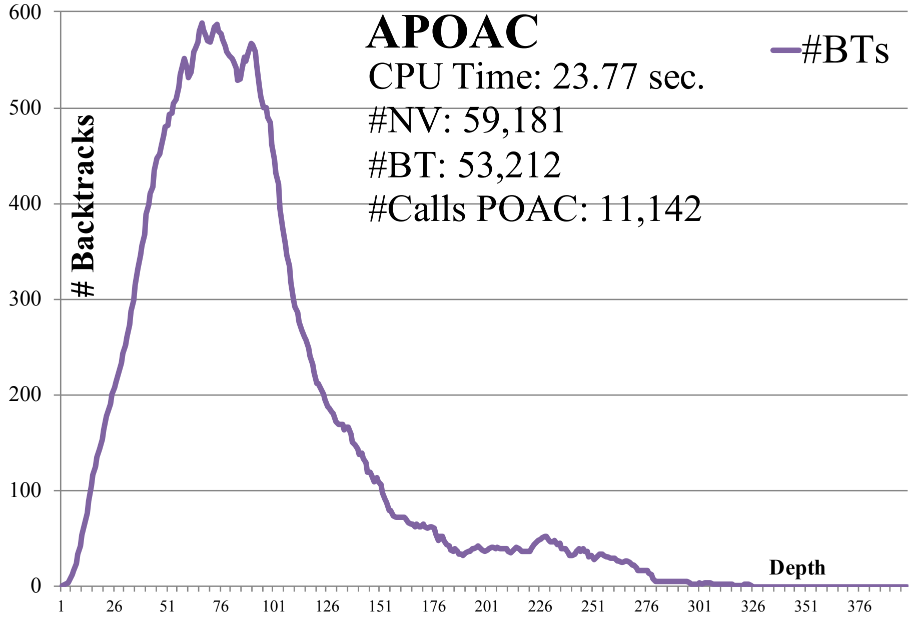


Figure 1.5: Number of backtracks per depth (BpD) using APOAC as an HLC for solving problem instance pseudo-aim-200-1-6-4

Moreover, in order to illustrate the effectiveness of the consistency algorithm, we further split the CpD into three curves corresponding to:

1. Calls deemed to be extremely effective in that they prune an entire subtree and yielded backtracking
2. Calls that are not particularly effective in that they cause some pruning but do not cause a wipeout
3. Calls that are totally wasted in that they do not yield any filtering

By comparing the three CpD curves, we detect where a consistency algorithm is effective and where its efforts are wasted.

Further, the superimposition of the BpD curve and the three CpD curves provides a qualitative indication of the performance of search and of the effectiveness of a consistency algorithm. Figure 1.6 shows the superimposition of the BpD and the three CpD curves for solving the problem instance pseudo-aim-200-1-6-4 from the pseudo-aim benchmark while enforcing APOAC. Note that:

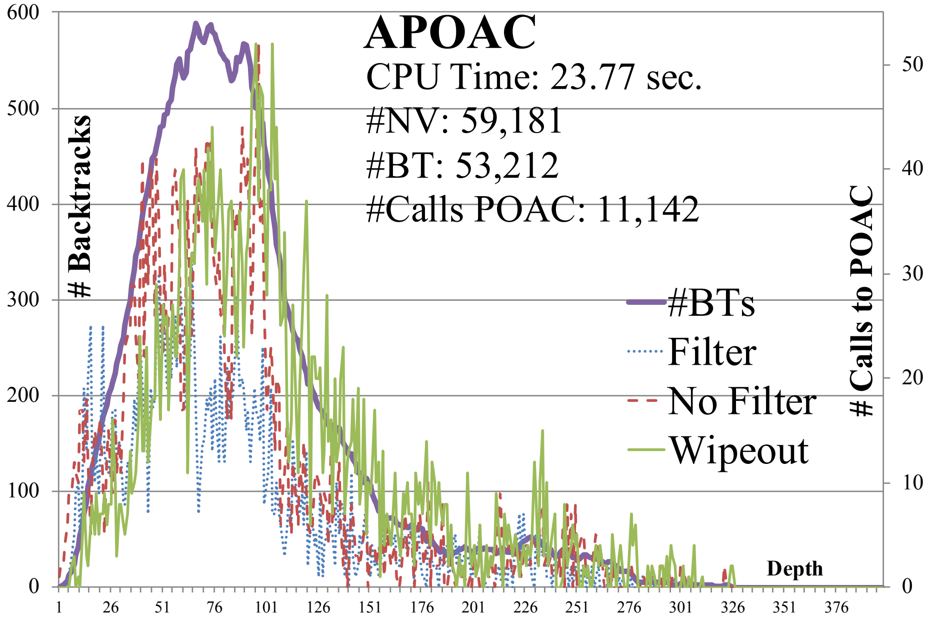


Figure 1.6: Superimposing the number of backtracks per depth (BpD) and the three types of number of calls per depth (CpD) to APOAC as an HLC for problem instance pseudo-aim-200-1-6-4

- The number of backtracks is shown on the left vertical axis.
- The number of calls to HLC (i.e., POAC) is shown on the right vertical axis.
- The purple line shows the number of backtracks per depth (BpD).
- The green line shows the number of HLC calls that are ‘extremely effective’ (i.e., yield wipeouts).
- The blue line shows the number of HLC calls that are ‘not particularly effective’ (i.e., filtering but no wipeouts).
- The red line shows the number of HLC calls that are ‘a total waste of effort’ (i.e., no filtering at all).

In Figure 1.6, we see that only one third of the calls to APOAC (i.e., the green curve) are really effective, which hints to the possibility of improving performance of search by 'firing' APOAC only when it is really effective.

We claim that *this visualization is a powerful explanation tool of the performance of search and effectiveness of an HLC and that it could even be used to allow a human user to directly intervene in the search process.*

1.2.2 'When:' Reactive Strategies for Enforcing HLC

In Constraint Processing, it is customary today to enforce the consistency known as Generalized Arc Consistency (GAC) at every step of the search process. As long as GAC allows search to effectively advance to deeper levels in the (tree-shaped) search space, GAC should remain the default consistency enforced. However, as thrashing occurs, we advocate to enforce stronger consistencies in order to more aggressively prune the search space and, subsequently, reduce the search effort. We propose to watch the number of backtrack steps during search as an indication of thrashing. To this end, we investigate three techniques: BTWatch, PrePeak, and PP-BTWatch:

1. BTWatch watches the number of backtrack along the search and triggers HLC whenever the counter reaches a given value, regardless of the position in the search tree.

2. PrePeak watches the number of backtracks per level of search (which is equal to the number of variables of the CSP) and enforces HLC at levels slightly shallower than the level where the peal value of the number of backtracks per level is observed.

3. PP-BTWatch is a hybrid between BTWatch and PrePeak, which watches the number of backtrack along the search and triggers HLC whenever the counter reaches a given value *and* the depth is before the level where a peak value of the number of backtracks per level is observed.

Further, in all three techniques, we use the same three geometric laws to update the value of the threshold for triggering HLC. The threshold value is updated in the following situations:

1. When HLC has been extremely effective (i.e., filtering yielded wipeout), we decrease the value of the threshold.
2. When HLC has not been particularly effective (i.e., some filtering but not wipeout), we slightly increase the value of the threshold.
3. When HLC was a total waste of effort (i.e., HLC resulted in no filtering at all), we aggressively increase the value of the threshold.

Overall, all three strategies are statistically equivalent, but exploring and evaluating them improves our understanding of reactive strategies.

1.2.3 ‘How Much:’ Monitoring Constraint Propagation

We explore three directions for monitoring the effectiveness of constraint propagation. First, enforce an ordering on the elements of the propagation queue of the HLC algorithm based on the activity of a variable/constraint or a structural property (e.g., elimination ordering). Second, because an HLC call can be costly in terms of time, we interrupt the execution of an HLC and allow it to process only a fraction of its propagation queue. Finally, we impose a bound on the duration of any call to HLC.

Combining PrePeak with the above three strategies yields PrePeak$^+$, which is the main contribution of this dissertation.

1.2.4 'Where:' Channel HLC along Cycles

Figure 1.7 shows a constraint network with two cycles intersecting on exactly one variable, which is an articulation node in the graph. This network is the 'poster child' to illustrate the importance of cycles. Indeed, instantiating the variable of the articulation node creates a chain, yielding a tractable CSP [Freuder, 1982]. More specifically, for this cycle, applying singleton arc consistency on the articulation node allows us to remove all values that do not participate in any solution (i.e., computes the minimal CSP). We theoretically characterize HLC properties that singleton-based consistencies guarantee (i.e., sufficient conditions) backtrack-free search on cactus and block graphs.

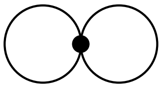


Figure 1.7: A constraint graph with two cyclic biconnected components

During search, we propose to exploit cycles in the constraint network of a CSP and channel constraint propagation along those cycles to improve the effectiveness of local consistency algorithms. In particular, we study two types of cycles in a constraint network, namely, a minimum cycle basis and triangles.

1.3 Contributions

In this section, we summarize our main contributions. We divide them into core contributions, which support the main claim of this dissertation, and secondary contributions, which are not directly related to the main thesis but are still valuable research results. Our core contributions are the following:

1. *A new visualization of the search effort* [Howell *et al.*, 2018b]. The proposed visualization tracks search progress and difficulties as well as the effort of enforcing consistency as a function of the depth of the search tree. This visualization has raised a sharp interest in discussions with the designers of several constraint solvers and is the topic of a new research direction in our laboratory.

2. *A reactive strategy for enforcing high-level consistency* [Woodward *et al.*, 2018]. We propose trigger-based strategies for enforcing high-level consistencies only when they are needed in order to exploit their effectiveness in pruning the search space while reducing the impact of the corresponding computational overhead. Further, we provide a unifying framework based on three orthogonal dimensions of 'when-where-how' to characterize how approaches for enforcing high-level consistency during search operate. Finally, we validate our approach for two HLCs, namely, POAC (a variable-based consistency property) and PC (a relational consistency property).

3. *New structural properties.* We identify new tractability results for block and cactus shaped constraint graphs. Exploiting our results about cactus graphs, we explore the benefits of channeling constraint propagation along cycles. More specifically, we propose to restrict POAC to the cycles of a minimum cycle basis of the graph [Woodward *et al.*, 2016a; Woodward *et al.*, 2017] and restrict path consistency to select triangles of the triangulated constraint graph. Future work should investigate exploiting our results for block graphs.

4. *A first practical algorithm for Partial Hyper-3 Consistency* (PH3C). We start

by investigating partial path consistency [Bliek and Sam-Haroud, 1999], empirically evaluating it as lookahead, which has never been studied. Jégou [1993] introduces, for non-binary CSPs, a relational-consistency property, called hyper-3 consistency (H3C), that is 'symmetrical' to path consistency for binary CSPs. The advantage of this property is that it allows us to operate on a special type of cycles, that is, triangles. We introduce a weakening of H3C into partial hyper-3 consistency (PH3C).[8] We introduce the first practical algorithm for enforcing PH3C during search. Importantly, we show that the 'dubois' benchmark[9] can be solved backtrack free using PH3C.

Our secondary contributions are the following:

1. *Weight-Based variable ordering in the context of a higher-level consistency* [Woodward and Choueiry, 2017]. Dom/wdeg is one of the most effective heuristics for dynamic variable ordering in backtrack search [Boussemart *et al.*, 2004]. As originally defined, this heuristic increments the weight of the constraint that causes a domain wipeout (i.e., a dead-end) when enforcing arc consistency during search. We explore alternatives for the weighing scheme in the context of two consistency properties, namely, POAC and RNIC.

2. *Adaptive parameterized consistency for non-binary CSPs by counting supports* [Woodward *et al.*, 2014]. Balafrej *et al.* [2013] proposed an adaptive parameterized consistency for binary CSPs as a strategy to dynamically select one of two local consistencies (i.e., AC and maxRPC). We propose a similar strategy for non-binary table constraints to select between enforcing GAC and pairwise consistency (PWC). This contribution is an instance of enforcing HLC only on

---

[8] Similar to the PPC algorithm for binary CSPs, PH3C operates on a triangulation of the dual graph of the CSP.

[9] Available from `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

select constraints, that is, along the axis ‘where’ of our proposed framework of ‘where-when-how much.’

3. *Witness-based search for solution counting* [Woodward *et al.*, 2016b]. Counting the exact number of solutions of a CSP is a difficult task (#P-complete) that is receiving increased attention in the research community. We propose witness-based search as a general improvement mechanism for any counting algorithm that exploits a tree decomposition of the CSP. and empirically establish the benefits of our technique in the context of two popular search-based counting algorithms.

1.4 Outline of Dissertation

The rest of this dissertation is organized as follows:

- **Chapter 2** reviews background information.
- **Chapter 3** introduces a novel way to visualize the search effort, which motivated this thesis. This contribution is at the source of a new research direction [Howell *et al.*, 2018b].
- **Chapter 4** introduces a strategy for dynamically enforcing higher-consistency by monitoring the performance search. Results from this chapter appeared in [Woodward *et al.*, 2018].
- **Chapter 5** discusses how to localize consistency properties and algorithms to operate on cycles in the graphical representation of a CSP. Preliminary results from this chapter appeared in [Woodward *et al.*, 2017; Woodward *et al.*, 2016a].

- **Chapter 6** discusses a special case of cycles, triangles, and enforcing Partial-Path Consistency and Partial Hyper-3 Consistency.

- **Chapter 7** concludes this dissertation and suggests directions for future research.

In order to maintain the coherence of this dissertation, incidental results and complementary information that are not central to the core contributions are organized in the appendices:

- **Appendix A** introduces weighting strategies for high-level consistency. Results from this chapter appeared in a technical report [Woodward and Choueiry, 2017].

- **Appendix B** introduces a method for adjusting the level of consistent by counting supports. Results from this chapter have been published [Woodward *et al.*, 2014].

- **Appendix C** introduces a scheme for improving the performance of solution counting by first finding a 'witness' solution in a sub-tree before counting all solutions. Results from this chapter appeared in a technical report [Woodward *et al.*, 2016b].

- **Appendix D** introduces how to determine the appropriate depth of search to attribute filtering when triggering higher-level consistency.

- **Appendix E** lists the benchmarks used in the experiments along with information regarding their hardness.

- **Appendix F** provides the details of the results of the experiments in Chapter 4.

Summary

This chapter introduced our motivation and claims, reviewed our approach and contributions, and described the structure of this dissertation.

Chapter 2

Background

In this chapter, we review background information about Constraint Satisfaction Problems (CSPs) useful for this dissertation. Then, we review the state of the art by casting previous approaches in terms of the three axes that we identified, namely, where, when, and how much.

2.1 Constraint Satisfaction Problem (CSP)

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

- $\mathcal{X}$ is a set of variables
- $\mathcal{D}$ is a set of domain values, where a variable $x_i \in \mathcal{X}$ has a finite domain $dom(x_i) \in \mathcal{D}$
- $\mathcal{C}$ is a set constraints restricting the combinations of values that can be assigned to the variables, where a constraint $c_i \in \mathcal{C}$ is defined by a scope $scope(c_i) \subseteq \mathcal{X}$ and a relation, which is a subset of the Cartesian product of the domains of the variables in $scope(c_i)$

A solution to the CSP assigns, to each variable, a value taken from its domain such that all the constraints are satisfied. The problem is to determine the existence of a solution and is $\mathcal{NP}$-complete.

Example 1 *Consider the Boolean CSP given by:*

- $\mathcal{X} = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N\}$
- $\mathcal{D} = \{D_A, D_B, D_C, D_E, D_F, D_G, D_H, D_I, D_J, D_K, D_L, D_M, D_N\}$, *where each* $D_i = \{0, 1\}$
- $\mathcal{C} = \{\langle R_1, \{ABCN\}\rangle, \langle R_2, \{IMN\}\rangle, \langle R_3, \{IJK\}\rangle, \langle R_4, \{AKL\}\rangle, \langle R_5, \{BDEF\}\rangle, \langle R_6\{CDH\}\rangle, \langle R_7, \{FGH\}\rangle, \langle R_8\{EFG\}\rangle\}$

The relations are given in the tables below:

R_1

A	B	C	N
0	0	0	1
0	1	0	1
0	1	1	0
1	0	1	1

R_2

I	M	N
0	1	0
1	0	1
1	1	1

R_3

I	J	K
0	0	0
0	1	1
1	1	0

R_4

A	K	L
0	1	0
1	1	1

R_5

B	D	E	F
0	0	0	1
0	1	0	1
1	0	1	0

R_6

C	D	H
0	0	0
1	0	0
1	1	0

R_7

F	G	H
0	0	0
1	0	0
1	1	0

R_8

E	F	G
0	0	0
1	0	0
1	1	0

The following variable-value pairs constitute a solution to this CSP:

$$\langle A, 0\rangle, \langle B, 1\rangle, \langle C, 1\rangle, \langle D, 0\rangle, \langle E, 1\rangle, \langle F, 0\rangle, \langle G, 0\rangle,$$

$$\langle H, 0\rangle, \langle I, 0\rangle, \langle J, 1\rangle, \langle K, 1\rangle, \langle L, 0\rangle, \langle M, 1\rangle, \langle N, 0\rangle.$$

The satisfying tuples are highlighted in the relations.

2.1.1 Solving a CSP

Backtrack search is a sound and complete method for finding a solution for a CSP [Bitner and Reingold, 1975]. In this dissertation, we do not use local search because it is not a complete algorithm and may miss a solution even when one exists.

Search operates by assigning a value to a variable and backtracks when a dead-end is encountered by undoing past assignments. The variable-ordering heuristic determines the order that variables are assigned in search, which can be dynamic (i.e., change during search). Boussemart *et al.* [2004] introduced dom/wdeg, a popular dynamic variable-ordering heuristic. This heuristic associates to each constraint $c \in \mathcal{C}$ a weight $w_c(c)$, initialized to one, that is incremented by one whenever the constraint causes a domain wipeout when enforcing arc consistency. The next variable x_i chosen by dom/wdeg is the one with the smallest ratio of current domain size to the weighted degree, $\alpha_{wdeg}(x_i)$, given by

$$\alpha_{wdeg}(x_i) = \sum_{(c \in \mathcal{C}_f) \wedge (x_i \in scope(C))} w_c(c) \tag{2.1}$$

where $\mathcal{C}_f \subseteq \mathcal{C}$ is the set of constraints with at least two future variables (i.e., variables who have not been assigned by search).

Modern solvers enforce a given consistency property on the CSP after each variable assignment. This lookahead removes from the domains of the unassigned variables values that cannot participate in a solution. Such filtering prunes from the search space fruitless subtrees, reducing the size of the search space and thrashing. The higher the consistency level enforced during lookahead, the stronger the pruning and the smaller the search space. A basic form of lookahead is *forward checking*, which filters the domains of only the unassigned variables connected, by a constraint, to the assigned variable. A more aggressive version of lookahead is *Real Full Lookahead*

(RFL) [Nadel, 1989], which enforces a given consistency property on the CSP induced by the unassigned variables (i.e., the future subproblem).

2.1.2 Representation

Several graphical representations of a CSP exist. Below, we introduce five graphical representations:

- In the *hypergraph*, the vertices represent the variables of the CSP, and the hyperedges represent the scopes of the constraints. Figure 2.1 shows the hypergraph of the CSP in Example 1.

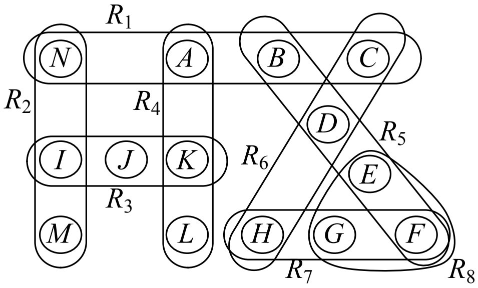


Figure 2.1: A hypergraph

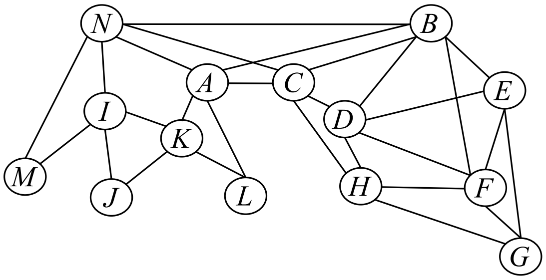


Figure 2.2: The primal graph

- In the *primal graph*, the vertices represent the CSP variables, and the edges connect every two variables that appear in the scope of some constraint. Figure 2.2 shows the primal graph of CSP whose hypergraph is shown in Figure 2.1.
- The *dual graph* is the graphical representation of the dual encoding of a CSP. The dual encoding of a CSP $\mathcal{P}$ is a binary CSP, $\mathcal{P}^D$, where the variables are the relations of $\mathcal{P}$, and their domains are the tuples of those relations. A constraint exists between two variables in $\mathcal{P}^D$ if their corresponding relations' scopes intersect. This constraint enforces the equality of the shared variables. Figure 2.3 shows the dual graph in Figure 2.1.

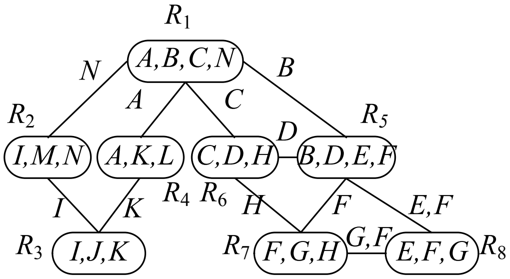


Figure 2.3: A dual graph

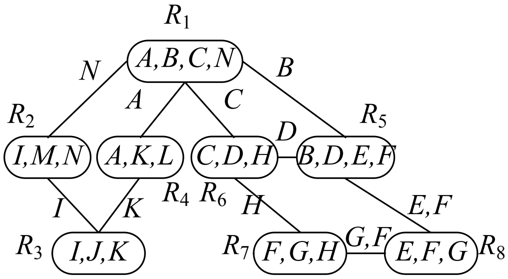


Figure 2.4: A minimal dual graph

- A *minimal dual graph* of a CSP is its dual graph with no redundant edges are removed. In the dual graph, an edge between two vertices is redundant if there exists an alternate path between the two vertices such that the shared variables appear in every edge in the path [Janssen *et al.*, 1989; Dechter, 2003a]. Redundant edges can be removed without changing the set of solutions. A minimal dual graph can be efficiently computed [Janssen *et al.*, 1989], but is not unique. Figure 2.4 shows a minimal dual graph of Figure 2.3 where the edge linking R_5 and R_7 is redundant, and thus removed.

- The *incidence graph* of a CSP is a bipartite graph where one set of vertices contains the variables of the CSP and the other set the constraints. An edge connects a variable and constraint if and only if the variable appears in the scope of the constraint. The incidence graph is the same graph used in the hidden-variable encoding [Rossi *et al.*, 1990]. Figure 2.5 shows the incidence of the CSP of Example 1.

2.1.3 Elimination Ordering and Graph Triangulation

An ordering of a graph is a total ordering of its vertices. The parents of a vertex are the neighbors that appear before it in the ordering. The width of a vertex is the number of its parents. The *width of an ordering* is the maximum vertex width. The

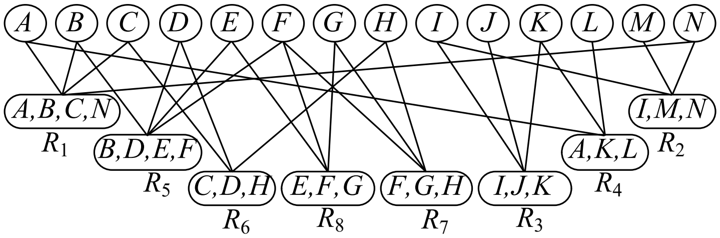


Figure 2.5: A incidence graph

width of a graph, denoted w, is the minimum width of all its possible orderings, and can be found in quadratic time in the number of vertices in the graph [Freuder, 1982].

A graph is *triangulated*, or *chordal*, iff every cycle of length four or more in the graph has a chord, which is an edge between two non-consecutive vertices. Graph triangulation adds an edge (a chord) between two non-adjacent vertices in every cycle of length four or more. While minimizing the number of edges added by the triangulation process is NP-hard, MinFill is an efficient heuristic commonly used for this purpose [Kjærulff, 1990; Dechter, 2003a]. Roughly, MinFill operates by determining, for each vertex, the number of edges needed to fully connect its parents (e.g., number of fill edges). It selects the vertex with the minimum number of fill edges and connects all of its parents. It then repeats until all the vertices have been selected.

A *perfect elimination ordering* of a graph is an ordering of the vertices such that, for each vertex v, v and the neighbors of v that occur after v in the ordering form a clique. If a graph is triangulated iff the graph has a perfect elimination ordering [Fulkerson and Gross, 1965]. The width of a triangulated graph is called the induced width, denoted w^*, of the ordering used.

2.1.4 Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of its constraint network. It is defined by a triple $\langle \mathcal{T}, \chi, \psi \rangle$, where $\mathcal{T}$ is a tree, and χ and ψ are two functions that determine which CSP variables and constraints appear in which nodes of the tree. The tree nodes are *clusters* of variables and constraints from the CSP. The set of variables of a cluster cl is denoted $\chi(cl) \subseteq \mathcal{X}$, and the set of constraints $\psi(cl) \subseteq \mathcal{C}$. A tree decomposition must satisfy two conditions:

1. Each constraint appears in at least one cluster and the variables in its scope must appear in this cluster; and
2. For every variable, the clusters where the variable appears induce a connected subtree.

Many techniques for generating a tree decomposition of a CSP exist [Dechter and Pearl, 1989; Jeavons *et al.*, 1994; Gottlob *et al.*, 2000]. We use here the tree-clustering technique [Dechter and Pearl, 1989].

1. First, we triangulate the primal graph of the CSP using the min-fill heuristic [Kjærulff, 1990].
2. Using the perfect elimination ordering given by the MAXCARDINALITY algorithm [Tarjan and Yannakakis, 1984], we identify the maximal cliques in the resulting chordal graph using the MAXCLIQUES algorithm [Golumbic, 1980], and use the identified maximal cliques to form the clusters of the tree decomposition. Figure 2.6 shows a triangulated primal graph of the example in Figure 2.1.

   The dotted edges (B,H) and (A,I) in Figure 2.6 are fill-in edges generated by the triangulation algorithm. The ten maximal cliques of the triangulated graph are highlighted with ‘blobs.’

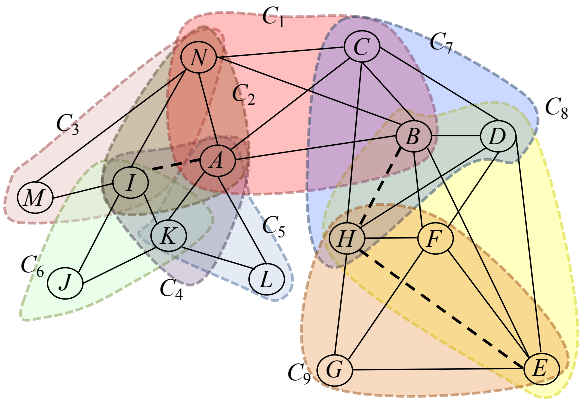


Figure 2.6: Triangulated primal graph and its maximal cliques

3. We build the tree by connecting the clusters using the JoinTree algorithm [Dechter, 2003a]. While any cluster can be chosen as the root of the tree, we choose the cluster that minimizes the longest chain from the root to a leaf.

4. Finally, we determine the variables and constraints of each cluster as follows: *a*) The variables of a cluster *cl*, $\chi(cl)$, are the variables in the maximal clique that yields the cluster; and *b*) The constraints of a cluster *cl*, $\psi(cl)$, are all the constraints R_i, such that $scope(R_i) \subseteq \chi(cl)$. Figure 2.7 shows a tree decomposition for the example of Figure 2.1. Note that we may end up with clusters with no constraints (e.g., C_2, C_4 and C_8).

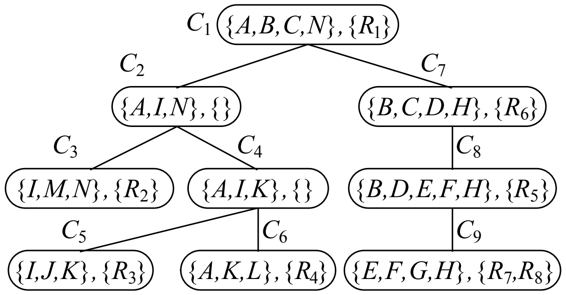


Figure 2.7: A tree decomposition of the CSP in Figure 2.1

A *separator* of two adjacent clusters is the set of variables that are associated with both clusters.

2.2 Consistency Properties and Algorithms

We distinguish between global and local consistency properties. Algorithms for enforcing a given consistency property typically operate by filtering values from the variables' domains or tuples from the constraints' relations. For any consistency property, there could be a number of algorithms for enforcing it on a CSP.

Global consistency properties are defined over the entire CSP. Minimality and decomposability are two global consistency properties [Montanari, 1974]. Constraint minimality requires that every tuple in a constraint appears in a solution. Decomposability guarantees that every consistent partial solution of any length can be extended to a complete solution. Decomposability is a highly desirable property: it guarantees that the CSP can be solved in a backtrack-free manner. Because guaranteeing a globally consistent CSP is in general exponential in time and space [Bessiere, 2006], we focus in practice on local consistency properties, which are in general tractable.

Local consistency properties are defined over combinations of a fixed size of variables (i.e., variable-based consistency) or constraints (i.e., relation-based consistency). A local consistency property guarantees that the values of all combinations of a given number of CSP variables (alternatively, the tuples of all combinations of a given size of CSP relations) are consistent with the constraints that apply to them. This condition is necessary but not sufficient for the values (or the tuples) to appear in a solution to the CSP.

Below, we review the main variable-based and relation-based consistency properties relevant to this dissertation.

2.2.1 Variable-Based Consistency

The most common property is Arc Consistency (AC) for binary CSPs, or Generalized Arc Consistency (GAC) for non-binary CSPs [Mackworth, 1977].

Definition 1 Generalized Arc Consistent (GAC) [Mackworth, 1977]*: A CSP is Generalized Arc Consistent (GAC) iff, for every constraint* c_i*, and every variable* $x \in \mathit{scope}(c_i)$*, every value* $v \in \mathit{dom}(x)$ *is consistent with* c_i *(i.e., appears in some consistent tuple of* R_i*).*

Algorithms for enforcing GAC remove domain values that have no GAC-support, leaving the relations unchanged [Bessière *et al.*, 2005]. Simple Tabular Reduction (STR) algorithms not only enforce GAC on the domains, but also remove all tuples $\tau \in R_j$ where $\exists x_i \in \mathit{scope}(R_j)$ such that $\tau[x_i] \notin \mathit{dom}(x_i)$ [Ullmann, 2007; Lecoutre, 2011; Lecoutre *et al.*, 2012].

Definition 2 Max Restricted Path Consistent (maxRPC) [Debruyne and Bessière, 1997a]: *A binary CSP is max Restricted Path Consistent (maxRPC) iff it is* $(1,1)$*-consistent and for all* $x_i \in \mathcal{X}$*, for all* $a \in \mathit{dom}(x_i)$*, for all* $x_j \in \mathcal{X}$ *s.t. there exists* $c \in \mathcal{C}$ *with* $\mathit{scope}(c) = \{x_i, x_j\}$*, there exists* b *in* $\mathit{dom}(x_j)$*, s.t. for all* $x_l \in \mathcal{X}$*, there exists* $d \in \mathit{dom}(x_l)$ *s.t. the 3-tuple* $((x_i, a), (x_j, b), (x_l, d))$ *is consistent.*

Informally, a problem is maxRPC iff it is $(1,1)$-consistent and for each value (x_i, a) and variable x_j linked to x_i by some constraint, there is a consistent extension b of a on x_j and this pair of values is path consistent.[1]

An extension of maxRPC to non-binary CSPs is maxRPWC.

Definition 3 max Restricted Pairwise Consistent (maxRPWC) [Bessière *et al.*, 2008]: *A CSP is max Restricted Pairwise Consistent (maxRPWC) iff* $\forall x_i \in \mathcal{X}$ *and* $\forall a \in$

---

[1] See Definition 10 for path consistency.

$dom(x_i)$, $\forall c_j \in \mathcal{C}$, where $xi \in scope(c_j)$, $\exists \tau \in rel(c_j)$ such that $\tau[x_i] = a$, τ is valid, and $\forall c_l \in \mathcal{C}(c_l \neq c_j)$, s.t. $scope(c_j) \cap scope(c_l) \neq \emptyset$, $\exists \tau \notin rel(c_l)$, s.t. $\tau[scope(c_j) \cap scope(c_l)] = \tau'[scope(c_j) \cup scope(c_l)]$ and τ' is valid. In this case we say that τ' is a PW-support of τ.

Singleton Arc-Consistency (SAC) ensures that no domain becomes empty when enforcing GAC after assigning a value to a variable [Debruyne and Bessière, 1997b]. This operation is called a *singleton test*. Let $\text{GAC}(\mathcal{P} \cup \{x_i \leftarrow v_i\})$ be the CSP after assigning $x_i \leftarrow v_i$ and running GAC.

Definition 4 Singleton Arc-Consistency (SAC) [Debruyne and Bessière, 1997b]: *A variable-value pair (x_i, v_i) of the CSP $\mathcal{P}$ is Singleton Arc-Consistency (SAC) iff* $\text{GAC}(\mathcal{P} \cup \{x_i \leftarrow v_i\}) \neq \emptyset$ *(the singleton check). $\mathcal{P}$ is SAC iff every variable-value pair is SAC.*

Algorithms for enforcing SAC remove all domain values that fail the singleton test.

Neighborhood SAC (NSAC) [Wallace, 2015] restrict the AC check of SAC to the neighborhood of a variable. Given a CSP $\mathcal{P}$ and $\mathcal{V}$ a subset of the variables of $\mathcal{P}$, we denote $\mathcal{P}|_{\mathcal{V}}$ the subproblem induced by $\mathcal{V}$ on $\mathcal{P}$. The constraints included in $\mathcal{P}|_{\mathcal{V}}$ are all those constraints whose scope contains a variable in $\mathcal{V}$.

Definition 5 Neighborhood Singleton Arc-Consistency (NSAC) [Wallace, 2015]: *A variable-value pair (x_i, v_i) of the CSP $\mathcal{P}$ is Neighborhood Singleton Arc-Consistency (NSAC) iff* $\text{GAC}(\mathcal{P}|_{\{x_i\} \cup neigh(x_i)} \cup \{x_i \leftarrow v_i\}) \neq \emptyset$ *(the singleton check). $\mathcal{P}$ is NSAC iff every variable-value pair is SAC.*

Partition-One Arc-Consistency (POAC) adds an additional condition to SAC [Bennaceur and Affane, 2001]. Let (x_i, v_i) denote a variable-value pair, $(x_i, v_i) \in \mathcal{P}$ iff $v_i \in dom(x_i)$.

Definition 6 Partition-One Arc-Consistent (POAC) [Bennaceur and Affane, 2001]: *A constraint network* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is Partition-One Arc-Consistent (POAC) iff* $\mathcal{P}$ *is SAC and for all* $x_i \in \mathcal{X}$*, for all* $v_i \in dom(x_i)$*, for all* $x_j \in X$*, there exists* $v_j \in dom(x_j)$ *such that* $(x_i, v_i) \in \text{GAC}(\mathcal{P} \cup \{x_j \leftarrow v_j\})$*.*

Balafrej *et al.* [2014] introduced two algorithms for enforcing POAC: POAC-1 and its adaptive version APOAC.

1. POAC-1 operates by enforcing SAC. In POAC-1, all the CSP variables are singleton tested and the process is repeated over all the variables until a fixpoint is reached. When running a singleton test on each of the values in the domain of a given variable, POAC-1 maintains a counter for each value in the domain of the remaining variables to determine whether or not the corresponding value was removed by any of the singleton tests. Values that are removed by each of those singleton tests are identified as not POAC and removed from their respective domains. POAC-1 typically reaches quiescence faster than SAC.

2. In APOAC, the adaptive version of POAC-1, the process is interrupted as soon as a given number of variables is processed. This number depends on input parameters and is updated by learning during search.

Neighborhood Inverse Consistency (NIC) [Freuder and Elfe, 1996] ensures that every value in the domain of a variable x_i can be extended to a solution of the subproblem induced by x_i and the variables in its neighborhood.

Definition 7 Neighborhood Inverse Consistency (NIC) [Freuder and Elfe, 1996]: *A variable* x_i *is Neighborhood Inverse Consistency (NIC) iff every value in* $dom(x_i)$ *can be extended to the variables in* $neigh(x_i)$ *that satisfies all the constraints in* $neigh(x_i)$*. A network is NIC iff every variable is NIC.*

2.2.2 Relation-Based Consistency

In the dual graph of a CSP, the vertices represent the CSP constraints and the edges connect vertices representing constraints whose scopes overlap. Relational Neighborhood Inverse Consistency (RNIC) [Woodward *et al.*, 2011b] enforces NIC on the dual graph of the CSP. That is, it ensures that any tuple in any relation can be extended in a consistent assignment to all the relations in its neighborhood in the dual graph.

Definition 8 Relational Neighborhood Inverse Consistent (RNIC) [Woodward *et al.*, 2011b]: *A relation* R_i *is Relational Neighborhood Inverse Consistent (RNIC) iff every tuple in* R_i *can be extended to the variables in* $\bigcup_{R_j \in \text{Neigh}(R_i)} \mathit{scope}(R_j) \setminus \mathit{scope}(R_i)$ *in an assignment that simultaneously satisfies all the relations in* $\text{Neigh}(R_i)$. *A network is* RNIC *iff every relation is* RNIC.

NIC and RNIC are theoretically incomparable [Woodward *et al.*, 2012], but RNIC has two main advantages over NIC:

1. NIC was originally proposed for binary CSPs and the neighborhoods in NIC likely grow too large on non-binary CSPs.

2. RNIC can operate on different dual graph structures to save time and/or improve propagation. Three variations of RNIC were introduced and operate on dual graphs that are minimal (wRNIC), triangulated (triRNIC), or both minimal and triangulated (wtriRNIC) [Woodward *et al.*, 2011a; Woodward *et al.*, 2011c]. Given an instance, selRNIC uses a decision tree to automatically select the dual graph for RNIC to operate on.

Definition 9 m-wise consistent [Gyssens, 1986; Janssen *et al.*, 1989]: *A CSP is* m*-wise consistent if, every tuple in a relation can be extended to every combination of* $m-1$ *other relations in a consistent manner.*

Pairwise Consistency (PWC) guarantees that every tuple consistent with a constraint c_i is consistent with every constraint in $neigh(c_i)$ [Gyssens, 1986]. Pairwise Consistency is equivalent to 2-wise consistency. Keeping with relational-consistency notations, Karakashian *et al.* denoted m-wise consistency by R($*$,m)C, and proposed a first practical algorithm for enforcing it [2010]. For simplicity, we will refer to R($*$,m)C as the property combining both GAC and R($*$,m)C, which can be obtained algorithmically by projecting the relations onto their scopes individually after enforcing R($*$,m)C.

Montanari [1974] originally introduced the property of path consistency as a tractable approximation of minimality.

Definition 10 Path Consistent (PC) [Dechter, 2003a]: *Given a CSP $\mathcal{P}$, the variables x_i and x_j are Path Consistent (PC) relative to a variable $x_{k \neq i}$ for every consistent assignment $\{(x_i, a), (x_j, b)\}$ there is some value $c \in dom(x_k)$ such that both the assignments $\{(x_i, a), (x_k, c)\}$ and $\{(x_j, b), (x_k, c)\}$ are consistent. $\mathcal{P}$ is path consistent iff $\forall x_i, x_j, x_k \in \mathcal{V}$ with $x_k \neq x_i \neq x_j$, x_i and x_j are path consistent relative to x_k.*

Directional path consistency (DPC) is a restriction of path consistency to an ordering *ord* of the variables, typically the perfect elimination ordering.

Definition 11 Directional Path Consistent (DPC) [Dechter and Pearl, 1988]: *A CSP is Directional Path Consistent (DPC) relative to order $ord = (x_1, x_2 \ldots, x_n)$, iff for every $k \geq i, j$, the two variables x_i and x_j are path consistent relative to x_k.*

Conservative Path Consistency (CPC) is a restriction of path consistency to the existing constraints of a problem. If there is no $C_{i,j} \in \mathcal{C}$ then x_i and x_j are conservative path consistent, otherwise x_i and x_j must be path consistent.

Definition 12 Conservative Path Consistent (CPC) [Debruyne, 1999]: *An assignment to two variables* x_i *and* x_j *such that there is no constraint* $C_{i,j} \in \mathcal{C}$ *is Conservative Path Consistent (CPC). If* $C_{i,j} \in \mathcal{C}$*, the assignment* $\{(x_i, a), (x_j, b)\}$ *is conservative path consistent iff* $(a, b) \in R_{i,j}$ *and* $\forall x_i, x_j, x_k \in \mathcal{V}$ *with* $k \neq i \neq j, C_{i,k}, C_{j,k} \in \mathcal{C} \Rightarrow \exists c \in dom(x_k)$ *such that* $(a, c) \in R_{i,k}$ *and* $(b, c) \in R_{j,k}$*. A constraint* $C_{i,j} \in \mathcal{C}$ *is conservative path consistent iff for all the tuples* $(a, b) \in R_{i,j}$*, the assignment* $\{(x_i, a), (x_j, b)\}$ *is conservative path consistent. A CSP is conservative path consistent iff it is arc consistent and* $\forall C_{i,j} \in \mathcal{C}$*,* $C_{i,j}$ *is conservative path consistent.*

Partial path consistency [Bliek and Sam-Haroud, 1999] was introduced in the same year as CPC. We present the definition as phrased by Lecoutre *et al.* [2011].

Definition 13 Partial Path Consistent (PPC) [Lecoutre *et al.*, 2011]*: A CSP is Partial Path Consistent (PPC) iff every closed graph-path of its constraint graph is consistent.*

The algorithms for enforcing PPC on a CSP involves triangulating the CSP (i.e., generating constraints for the added triangulated edges) and enforcing CPC on the triangulated network.

Definition 14 Conservative Dual Consistent (CDC) [Lecoutre *et al.*, 2007]: *Given a CSP,* $\mathcal{P}$*, an assignment* $\{(x_i, a), (x_j, b)\}$ *is Conservative Dual Consistent (CDC) iff* $(c_{i,j} \notin \mathcal{C}) \vee ((x_j, b) \in AC(\mathcal{P}|_{x_i \leftarrow a}) \wedge (x_i, a) \in AC(\mathcal{P}|_{x_j \leftarrow b}))$*.* $\mathcal{P}$ *is conservative dual consistent iff every consistent assignment* $\{(x_i, a), (x_j, b)\}$ *is conservative dual consistent.*

CDC combined with AC is called *Strong* Conservative Dual Consistency (sCDC).

2.2.3 Comparing Consistency Properties

Using the terminology of Debruyne and Bessière [1997b], we say that a consistency property p is *stronger* than p' if in any CSP where p holds p' also holds. Further, we say that p is *strictly stronger* than p' if p is stronger than p', and there exists at least one CSP in which p' holds but p does not. We say that p and p' are equivalent if p is stronger than p', and vice versa. Finally, we say that p and p' are incomparable when there exists at least one CSP in which p holds but p' does not, and vice versa. In practice, when a consistency property p is stronger than another p', enforcing p never yields less pruning than enforcing p' on the same problem.

Following this terminology, POAC is strictly stronger than SAC, which is strictly stronger than GAC. The consistency property enforced by the adaptive algorithm APOAC is strictly stronger than GAC, incomparable with SAC, and strictly weaker than POAC.

Below, we introduce a new result.

Theorem 1 *On binary CSPs, Conservative Path Consistency (CPC) is equivalent to R(∗,3)C.*

Proof: By contradiction.

- $\Rightarrow$: Assume that CPC removes more tuples than R(∗,3)C does. Thus, CPC filters a tuple $\tau_{i,j}$ on variables i, j when trying to extend the tuple to a third variable k. Three constraints must exist between these three variables because of the conservative property. Thus, there is a combination of three constraints $c_{i,j}$, $c_{i,k}$, $c_{j,k}$ in R(∗,3)C. Thus, R(∗,3)C attempts to extend $\tau_{i,j}$ to a tuple in $c_{i,k}$ and a tuple in $c_{j,k}$, which filters $\tau_{i,j}$ because CPC filtered this tuple using the same combination, which yields a contradiction.

$\Leftarrow$: Assume that R($*$,3)C filters more than CPC. Thus, R($*$,3)C filters a tuple $\tau_{i,j}$ on constraint $c_{i,j}$ when extending to a combination of three constraints $c_{i,j}, c_2, c_3$ in the dual graph. We illustrate by contradiction that this cannot happen for all possible scopes for c_2 and c_3 on binary CSPs:

1. $\mathit{scope}(c_2) = \{i, k\}$ and $\mathit{scope}(c_3) = \{j, k\}$. Thus R($*$,3)C cannot extends $\tau_{i,j}$ to variable k. But, CPC would have filtered this tuple.
2. $\mathit{scope}(c_2) = \{v, k\}$ and $\mathit{scope}(c_3) = \{v, l\}$, where v is i or j. All dual constraints are an equality constraint over the common subscope v, the edge between c_2 and c_3 is redundant, which means R($*$,3)C cannot obtain filtering stronger than PWC, which on binary CSPs is equivalent to GAC. Thus, this tuple could not be removed.
3. $\mathit{scope}(c_2) = \{i, k\}$ and $\mathit{scope}(c_3) = \{j, l\}$. Thus, R($*$,3)C forms a chain of three constraints, which on binary CSPs is equivalent to GAC. Thus, the tuple could not be removed.

All situations yield a contradiction. ☐

2.3 Minimum Cycle Basis

A *cycle basis* of a graph is a maximal set of cycles that are linearly independent (i.e., cycles in the basis cannot be obtained by taking the composition of other cycles in the basis)[2] [Horton, 1987]. In a weighted graph, the weight of a cycle in the graph is the sum of the weights of the edges in the cycle. A *minimum cycle basis* (MCB) is a cycle basis where the sum of the weights of the cycles in the cycle basis is minimum. Informally, a minimum cycle basis is a minimum set of cycles that can generate all

---

[2] The composition of two cycles is the symmetric difference (exclusive-or) between the edges of the cycles.

the cycles of the graph. In the case of an unweighted graph, the weights of each edge is one, a minimum cycle basis has a minimum total length.[3] Algorithms for finding a minimum cycle basis are either exact or approximate, finding the minimum within some bound [Horton, 1987; Kavitha *et al.*, 2007; Mehlhorn and Michail, 2009; Amaldi *et al.*, 2010]. The complexity of the exact algorithm is $\mathcal{O}(e^2 n/\log(n))$ where n is the number of vertices and e the number of edges in the graph [Amaldi *et al.*, 2010]. That of the approximate algorithm is $\mathcal{O}(e^{\omega}\sqrt{n\log(n)})$ where ω is the best exponent of matrix multiplication ($\omega < 2.376$) [Kavitha *et al.*, 2007].

Figure 2.8 shows the incidence graph of Example 1), where circles denote the variables and the squares the constraints. The graph has thirteen cycles:

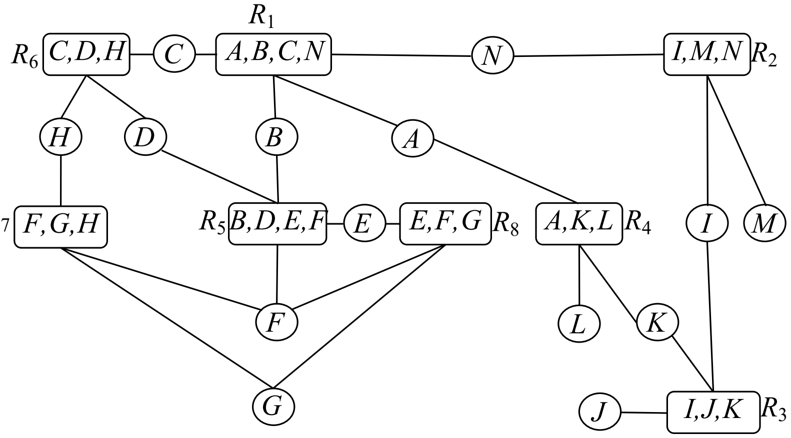


Figure 2.8: A re-arrangement of the incidence graph of Figure 2.5

1. (R_6, H, R_7, F, R_5, D)
2. (R_6, D, R_5, B, R_1, C)
3. (R_5, F, R_8, E)
4. (R_7, G, R_8, F)

---

[3] Note that an MCB is not unique.

5. $(R_1, A, R_4, K, R_3, I, R_2, N)$
6. $(R_6, H, R_7, F, R_5, B, R_1, C)$, obtained by the symmetric difference of first and second cycle.
7. $(R_6, H, R_7, F, R_8, E, R_5, D)$, obtained from the symmetric difference of first and third cycle.
8. $(R_6, H, R_7, G, R_8, F, R_5, D)$, obtained from the symmetric difference of first and forth cycle.
9. (R_5, F, R_7, G, R_8, E), obtained form symmetric difference of third and forth cycle.
10. $(R_6, H, R_7, F, R_8, E, R_5, B, R_1, C)$, obtained from the symmetric difference of first, second, and third cycle.
11. $(R_6, H, R_7, G, R_8, F, R_5, B, R_1, C)$, obtained from the symmetric difference of first, second, and forth cycle.
12. $(R_6, H, R_7, G, R_8, E, R_5, D)$, obtained from the symmetric difference of first, third, and forth cycle.
13. $(R_6, H, R_7, G, R_8, E, R_5, B, R_1, C)$, obtained from the symmetric difference of first, second, third, and forth cycle.

Notice the sixth through thirteenth cycle can be obtained from symmetric difference of the first five. Thus, the first five cycles constitute a minimal cycle basis for this graph. Incidentally, note that the variables M, J, L do not appear in any cycle.

2.4 Related Literature

We organize the related work along the three axes shown in Figure 2.9 and combinations of these dimensions.

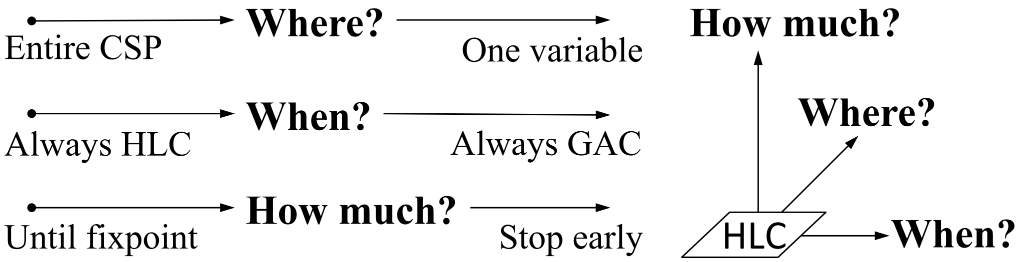


Figure 2.9: Dimensions of enforcing consistency

2.4.1 Where

The consistency level is chosen based on some property of the variables and/or constraints. One can exploit structural properties of the constraint network, such as the neighborhood of a variable or a constraint [Freuder and Elfe, 1996; Wallace, 2015; Woodward *et al.*, 2011b], or some configuration of constraints [Karakashian *et al.*, 2010]. Freuder and Wallace [1991] enforce arc consistency on a subproblem within a given distance (i.e., where) from the instantiated variable. Balafrej *et al.* [2013] and Woodward *et al.* [2014] exploit the degree of support that constraints provide to variable-value pairs, which is a structural property.

2.4.2 When

The consistency selected depends on search performance. Borrett *et al.* [1996] switch between backtrack algorithms, level of consistency enforced, and ordering heuristics by a complex combination of domain sizes, number of variables, and backtrack levels.

Epstein *et al.* [2005] consider several strengths of AC-based consistencies depending on the depth of the search tree. Balafrej *et al.* [2015] use a multi-armed bandit at each depth of search tree to select between MAC, maxRPC, or POAC.

2.4.3 How much

Propagation is terminated before reaching a fixpoint. Such approaches focus on the propagation queue of a consistency algorithm. They either order the propagation queue according to some heuristic [Wallace and Freuder, 1992] or interrupt the consistency algorithm when the pruning effect of propagation has subsided [Balafrej *et al.*, 2014] or the allocated time has elapsed [Eén and Biere, 2005; Geschwender *et al.*, 2016].

2.4.4 Where and when

Some authors propose heuristics to dynamically switch from GAC to a stronger property on a selection of constraints (i.e., where) based on the amount of activity of the constraints during search (i.e., when). For example, Stergiou [2008] switches between GAC and maxRPC for binary CSPs and Paparrizou and Stergiou [2012] between GAC and maxRPWC for nonbinary CSPs.

2.4.5 Where and how much

Paparrizou and Stergiou [2017] propose a strategy for interrupting enforcing Neighborhood-SAC based on the amount of filtering it yields. For each singleton test on the considered variable, the filtering is interrupted (i.e., how much) unless the domain of any neighboring variable (i.e., where) becomes singleton.

Summary

In this chapter, we gave background information on CSPs. We described representations of a CSP and introduced some common consistency properties and reviewed how they can be compared. Finally, we reviewed the main approaches to enforcing high-level consistency during while positioning each approach along the three orthogonal directions that we have identified, thus validating the relevance of our proposed characterization.

Chapter 3

Visualizing Search

Carro and Hermenegildo [1998] distinguish three main uses of visualization in Constraint Programming:

1. *Debugging*: Providing a clear view of the program state to the programmer.
2. *Tuning and optimizing programs*: Providing, to the programmer or the expert user, profiling statistics about the solver's execution.
3. *Teaching and education*: Providing explanations to a layperson.[1]

Thus, the design of any visualization takes into account the intended use, or the goal, of the visualization. In this chapter we focus the goal of 'tuning and optimizing programs.' To that end we introduce a visualization to help with understanding the performance of backtrack search and the effectiveness of enforcing a local consistency property on a problem instance.

Below, we first review previous approaches to visualization in Constraint Programming. Then, we introduce our proposed visualizations [Howell *et al.*, 2018b].

---

[1] For example, the visualization of constraint solving in the context of Sudoku (`http://sudoku.unl.edu`), Minesweeper (`http://minesweeper.unl.edu`), the Game of Set (`http://gameofset.unl.edu`), and SAT solving (`http://satviz.unl.edu`).

We discuss how our visualizations allow us to interpret the performance of search, to compare the performances of two or more search algorithms, and to understand the effectiveness of enforcing a particular local consistency during search. Finally, we discuss two aspects of our implementation: how to provide a real-time visualization of search and how to enforce multiple consistency algorithms during search.

3.1 Previous Approaches to Visualizing Search

In Constraint Programming, visualizations are developed for the search tree and for the constraints, which are typically global constraints. The visualization in this dissertation focuses on the former because we operate on arbitrary constraints. Prior research on search-tree visualization focuses on the 2-way branching scheme, which is typical in Constraint Programming, in contrast to the k-way branching scheme adopted by the CSP community. In the constraint solver CHIP, Simonis and Aggoun [2000] propose to visualize the search tree from two perspectives, namely, *tree view* and *phase-line display*:

- *Tree view:* The search tree is displayed using a parent-children relationship. Each node in the tree is an variable-assignment that was consistent after enforcing lookahead. Failed branches are ‘collapsed’ to keep the display of the tree manageable. Figure 3.1 shows an example tree view.

- *Phase-line display:* Each variable is given a line that shows the depth in the search tree at which the variable was assigned (y-axis) as time progresses (x-axis). This visualization would show horizontal lines for a static variable-ordering, and can be useful for visualizing dynamic variable-ordering heuristics. Figure 3.2 shows an example phase line display.

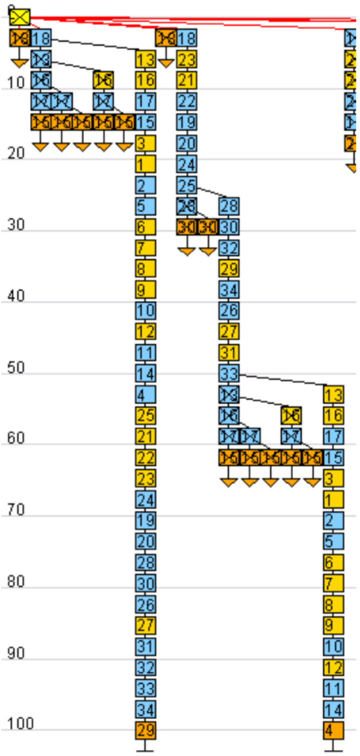


Figure 3.1: The tree view [Simonis and Aggoun, 2000]

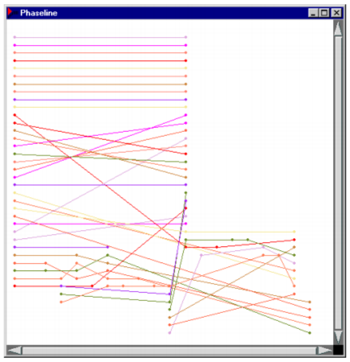


Figure 3.2: The phase-line display [Simonis and Aggoun, 2000]

In addition to these two views of the search tree, Simonis and Aggoun [2000] provide functionalities that allow an in-depth analysis of the states of the variables and constraints, and to view the order of the constraints considered during propagation. Simonis *et al.* [2000] also introduce visualizations of global constraints in the context of the constraints meaning in the CSP.

The tree view and phase-line display were originally proposed in the larger DiSCiPl project. The DiSCiPl project provides extensive visual functionalities to develop, test, and debug constraint logic programs such as displaying variables' states, effect of constraints and global constraints, and event propagation at each node of the search tree [Simonis and Aggoun, 2000; Carro and Hermenegildo, 2000]. Many useful methodologies from the DiSCiPl project are implemented in CP-VIZ [Simonis *et* *al.*, 2010] and other works [Shishmarev *et al.*, 2016]. The implementation of CP-Viz is solver-agnostic. It takes as input an XML trace of the solver and generates visualizations of that search.

The OZ Explorer displays the search tree allowing the user to access detailed information about the node at each tree node and to collapse and expand failing trees for closer examination [Schulte, 1996]. This work is currently incorporated into Gecode's Gist [Schulte *et al.*, 2015].

The above approaches focus on exploring the search tree (as well as a problem's components) while our work proposes particular projections (i.e., views, summaries) of the data *reflecting* (i.e., compiling) the cost and the effectiveness of both search and enforcing consistency. We believe that these visualizations are orthogonal and complementary.

Tracking search effort by depth was first proposed by Epstein *et al.* [2005] for the number of constraint checks and values removed per search and by Simonis *et al.* [2010] in CP-Viz for the number of nodes visited (also used for solving a packing problem [Simonis and O'Sullivan, 2011]). Figure 3.3 shows an example visualization of the number of constraint checks at every depth of search [Epstein *et al.*, 2005]. Figure 3.4 shows an example visualization of the result of every node visit call, either a failure (i.e., found the current subtree inconsistent) or successful (i.e., try a variable instantiation) [Simonis *et al.*, 2010].

We claim that the number of constraint checks, values removed, and nodes visited are not accurate measures of the thrashing effort. Indeed, the number of constraint checks varies with the degree of the variables. The number of values removed and nodes visited vary with the size of the domain. In contrast, we claim that the number of backtracks per search depth (BpD) provides a more faithful representation of the thrashing effort, which is exactly the aspect of search that we aim to characterize.

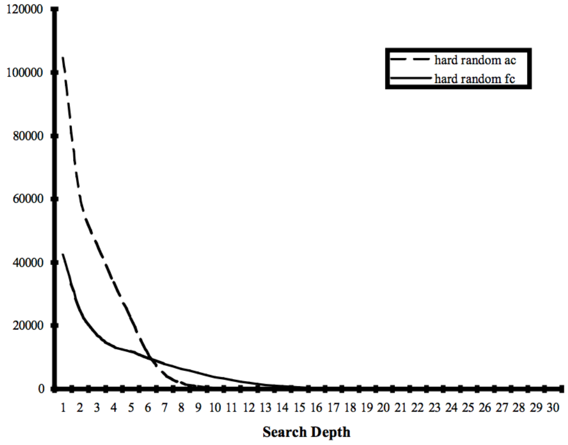


Figure 3.3: The number of each constraint check at every depth [Epstein *et al.*, 2005]

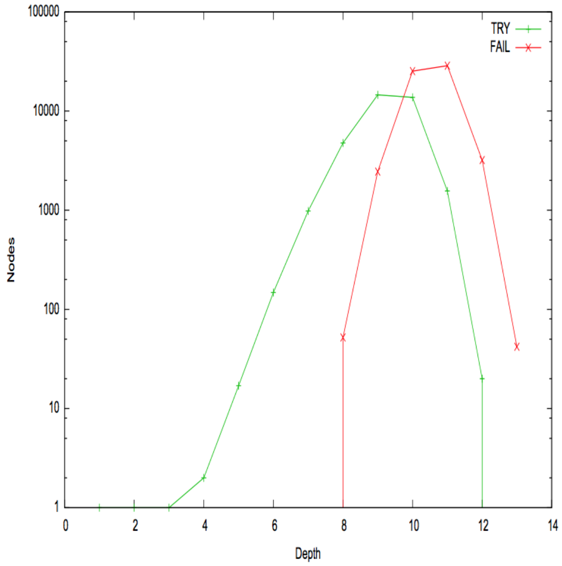


Figure 3.4: The result of a node visit at every depth [Simonis *et al.*, 2010]

Recently, techniques have appeared in Constraint Processing for dynamically choosing between a set of consistency properties based on the CPU time spent on exploring a given subtree [Balafrej *et al.*, 2015]. We claim that we better track the effectiveness of such decisions by following the number of backtracks per depth (BpD) and the number of consistency calls per depth (CpD) rather than the CPU time of searching a given subtree.

3.2 Analyzing Search Effectiveness

We propose two visualizations towards summarizing and explaining the performance of search:

1. We track the number of *backtracks per depth* (BpD) at each level of search to understand where and how search struggles and where it smoothly proceeds.

2. To understand the impact of enforcing a given consistency property, we track

the number of *calls to the consistency algorithm per depth* (CpD) in the search tree. Further, we split these calls into three categories: those that yield domain *wipeout* (i.e., detect inconsistency), those that effectively *filter* domains without detecting a dead-end, and those that yield *no filtering*.

3.2.1 Backtracks per Depth

The Backtracks per Depth (BpD) chart reflects various aspects of search effectiveness as we illustrate with an example. Table 3.1 reports runtime statistics of search for solving a coloring problem while enforcing the GAC algorithm STR2+ [Lecoutre, 2011] and POAC algorithm POAC-1 [Balafrej *et al.*, 2014] using the dom/wdeg ordering heuristic [Boussemart *et al.*, 2004].[2] The definitions of GAC (Definition 1 in Section 2.2.1) and POAC (Definition 6 in Section 2.2.1) are not needed for this discussion: it suffices to say that an algorithm that enforces GAC is generally quick but does little filtering while a POAC algorithm is typically costly but can prune larger substrees of the search space than the GAC algorithm. As we can see in Table 3.1, it is clear that our 'investment' in POAC is worthwhile because POAC solves the instance in about 41 minutes while GAC does not terminate.

Table 3.1: Search with GAC and POAC on 4-insertions-3-3. Note that GAC timed out

Algorithm	**CPU Time (sec)**	**# Nodes Visited**	**# Backtracks**	**max$_{\text{BpD}}$**
GAC	>8,099.9	335,498,250	243,259,300	15,241,175
POAC	2,447.4	1,325,469	930,208	59,756

Figure 3.5 shows the BpD charts of the search with GAC (left) and POAC (right). We see that GAC thrashes around depth 50 with max$_{\text{BpD}}$=15,241,175 backtrack at depth 53. POAC, which enforces a strictly stronger consistency throughout search,

---

[2] Instance 4-insertions-3-3 of the benchmark graphColoring-k-insertions from `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`.

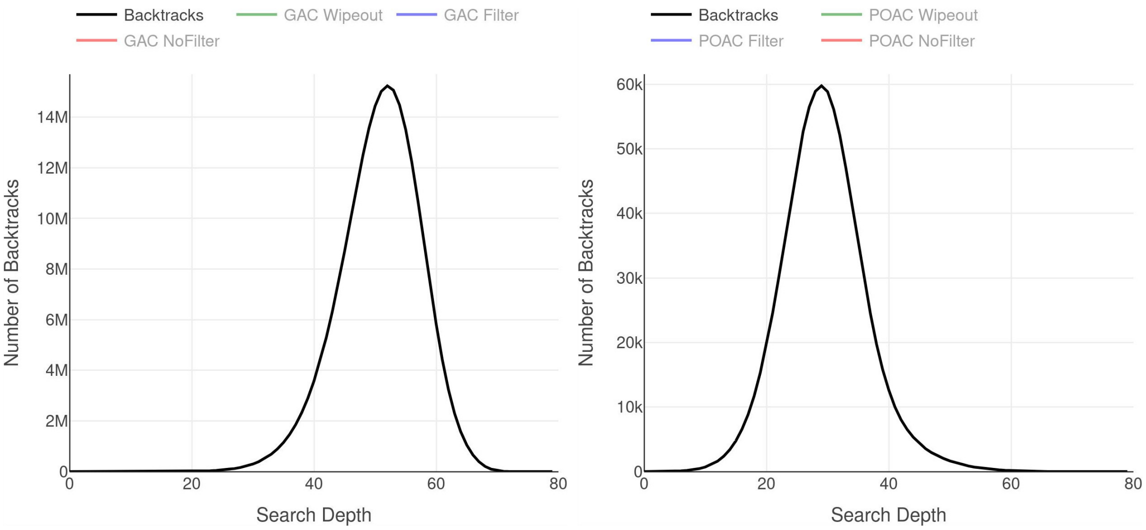


Figure 3.5: BpD for GAC (left) and POAC (right) on instance 4-INSERTIONS-3-3.

limits the severity of thrashing to only $\max_{\text{BpD}}$=59,756 backtracks at depth 29. By detecting and pruning inconsistencies at a shallower search level, POAC solves the problem while GAC fails.

3.2.2 Calls per Depth

Enforcing a higher-level consistency (HLC), such as POAC, after each variable instantiation during search is not always worthwhile. On easier problems, the computational cost of a HLC can be an overkill. We propose another visualization to examine the effectiveness of enforcing a high-level consistency by superimposing, to the BpD chart, the Calls per Depth (CpD) chart reporting the *number of calls to* POAC *per depth.* Figure 3.6, unsurprisingly shows that the BpD and the CpD charts of POAC largely overlap in shape (modulo their respective ranges shown on both sides of the chart), which is explained by the fact that POAC is called at every variable instantiation during search. In other dynamic strategies where two or more levels of consistency are enforced, the CpD would allow us to differentiate between the impact of each

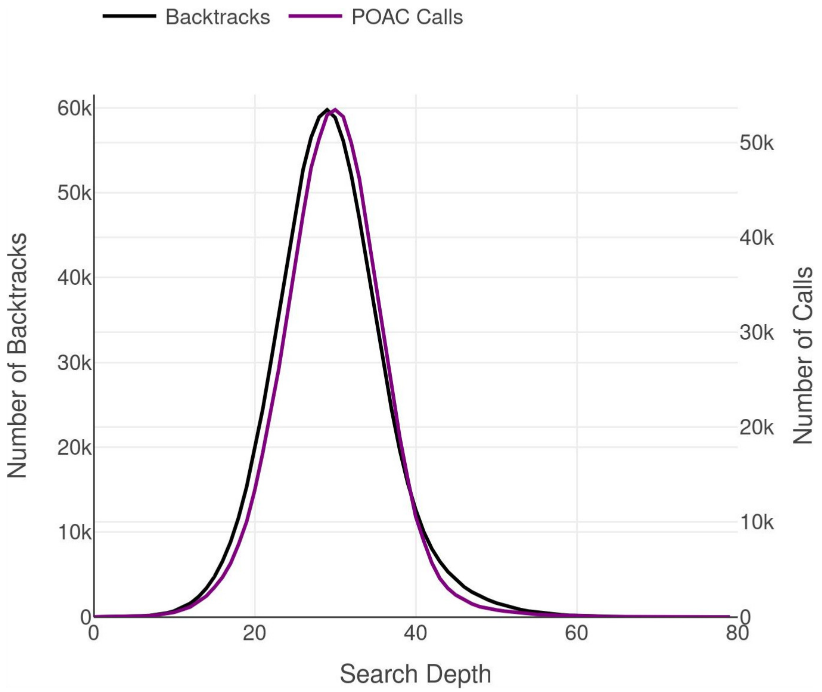


Figure 3.6: Superimposing CpD and BpD for POAC on 4-INSERTIONS-3-3

consistency algorithm.

We propose to split more finely the CpD into three categories depending on whether calls to HLC resulted in:

1. a domain wipeout (the most effective HLC calls, which cause backtracking),
2. filtering but no wipe out (which prunes inconsistent subtrees, reducing the search space, but cannot detect inconsistency), and
3. no filtering (which are wasteful calls to HLC).

In Figure 3.7, these three CpDs are shown in green, blue, and red, respectively. In the case of our particular example, we can see that the wasteful calls to POAC, shown in red, are extremely few and that almost all calls are effective (green or blue). This realization fully explains the ability of POAC to prevent search from thrashing at deeper search levels and its effectiveness in solving this difficult instance.

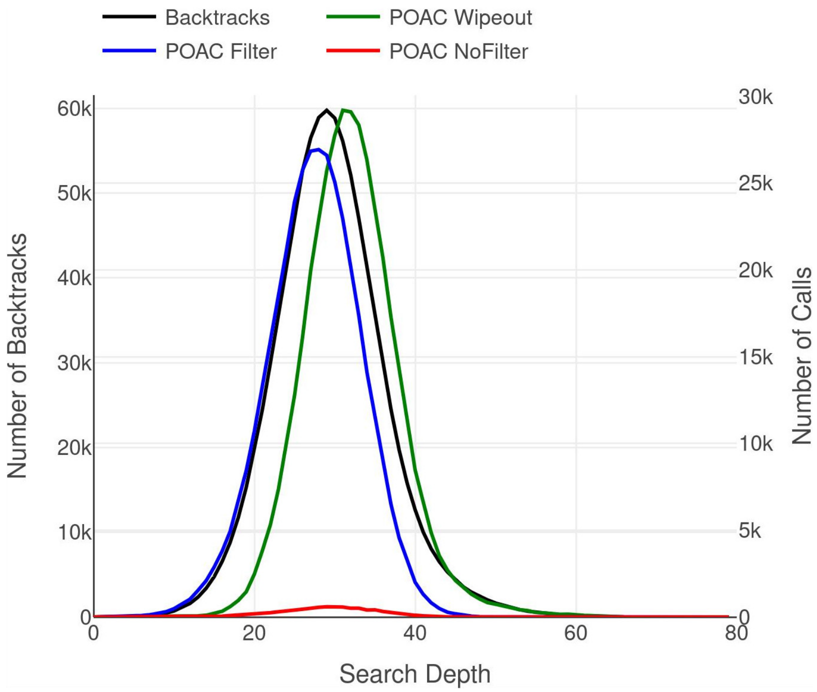


Figure 3.7: Superimposing BpD and detailed CpD (wipeout in green, filtering in blue, no-filtering in red) for POAC on 4-INSERTIONS-3-3

3.3 Comparing Different Consistency Algorithms

We use the BpD and CpD to understand and compare the behavior of PREPEAK$^+$, which is a new reactive strategy for enforcing high level consistency during search described in Chapter 4.

To this end, we solve the CSP instance PSEUDO-AIM-200-1-6-4 with backtrack search under three settings: (1) maintaining GAC, (2) with APOAC, and (3) PREPEAK$^+$. PREPEAK$^+$ is conservative in that it primarily enforces GAC. However, it triggers an HLC, such as POAC, when the number of backtracks per depth (BpD) reaches a given threshold value θ but only when search backtracks to levels shallower than the depth where the threshold is met. PREPEAK$^+$ keeps firing the HLC as long as the BpD at the considered depth is smaller than θ. Furthermore, every time it backtracks, PREPEAK$^+$ updates the values of θ by reducing it or increasing it according to three geometric laws depending on whether the HLC yields wipeout (i.e., it is effective), filters the search space, or yields no filtering (i.e., the HLC calls are wasteful).

Figure 3.8 shows the BpD for GAC at the end of search. This curve exhibits a peak at depth 92 with 34,023 backtracks at that depth, showing that GAC is too weak to filter out bad values: it spends much of its time thrashing around this depth level.

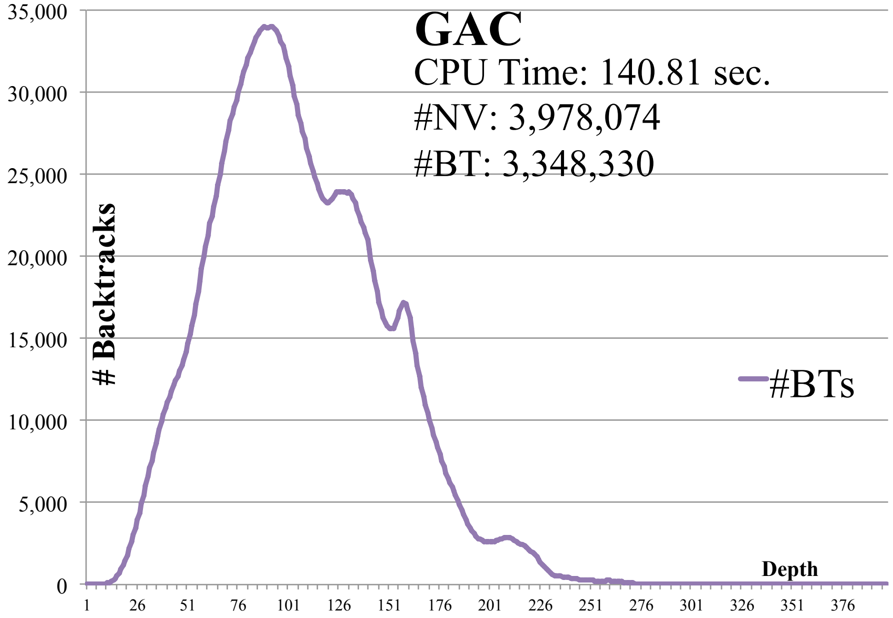


Figure 3.8: BpD and CpD of GAC on PSEUDO-AIM-200-1-6-4

Figure 3.9 shows the BpD (purple) and CpD (colored) curves for APOAC. Examining the BpD curve, we realize that APOAC so effectively prunes the 'bad subtrees' from the search space that it dramatically reduces the number of backtracks at the peak depth down to 407 and the location of peak to around depth 75. We see that this instance benefits from enforcing an HLC such as POAC with a clear benefit on the CPU time (which is reduced by one order of magnitude from GAC). However, by observing the colored curves in Figure 3.9, we notice that the number of calls

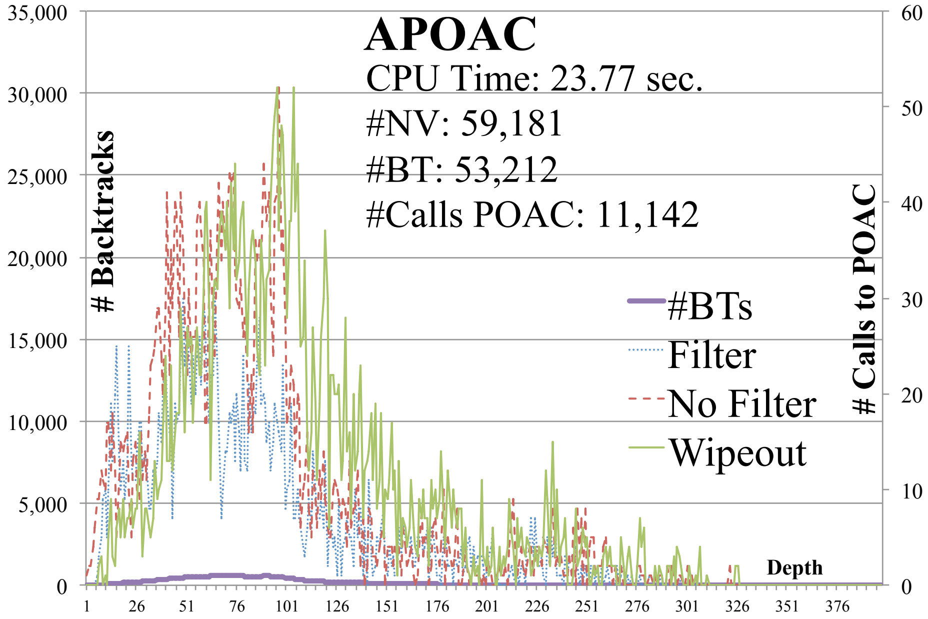


Figure 3.9: BpD (purple) and CpD's (colored) of APOAC on PSEUDO-AIM-200-1-6-4

to POAC that are ineffective (red curve) are of the same order as those that yield wipeout (green curve). The detailed CpD curves hint to some savings that could be further obtained could one cancel the wasteful calls to POAC.

Figure 3.10 shows the BpD (purple) and CpD (colored) curves for PREPEAK$^+$. PREPEAK$^+$ is conservative in that it calls an HLC only when search thrashes, justifying the cost of a stronger but more costly consistency algorithm. Indeed, we observe that the peak value of BpD is smaller than for GAC but greater than for APOAC (2,421 versus 34,023 and 407, respectively). However, examining the detailed CpD curves shows that advantage of PREPEAK$^+$: Indeed, the wasteful calls to POAC (red) are almost eliminated and the total calls to POAC are reduced down to 228 for PREPEAK$^+$ from 11,142 for APOAC. This economy in the calls to POAC is immediately translated by the reduction of the CPU time. Thus, despite the fact that PREPEAK$^+$ explores a larger search tree than APOAC (see number of nodes visited) because it does not call the HLC at each variable instantiation, it effectively reacts

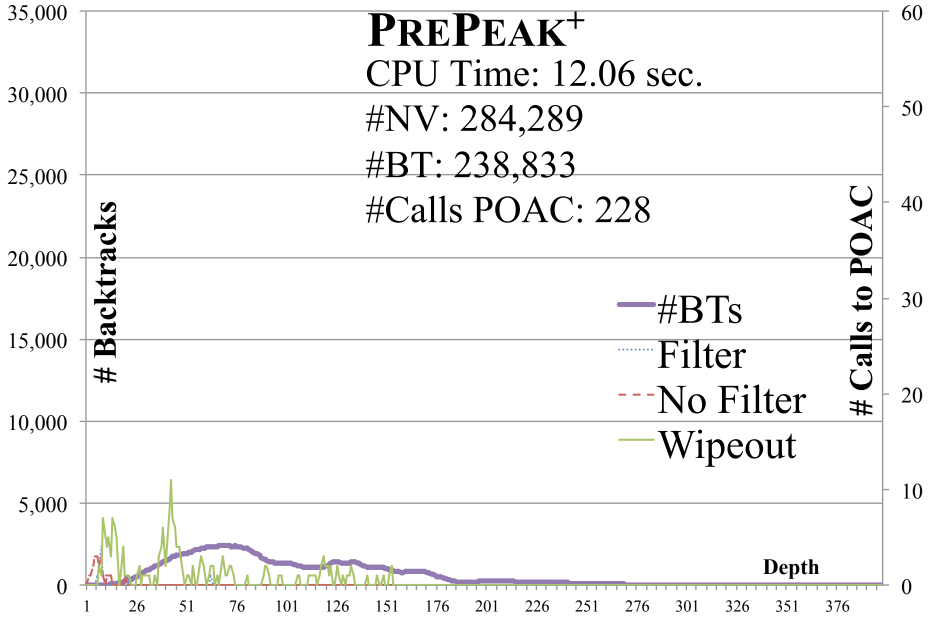


Figure 3.10: BpD (purple) and CpD's (colored) of PrePeak$^+$ on pseudo-aim-200-1-6-4

to thrashing, calling the HLC only when it is needed, but spontaneously reverting to GAC otherwise.

This example illustrates the pertinence of the tools provided by BpD and CpD in visually explaining the behavior of search and the benefits of PrePeak$^+$.

3.4 Implementing the Visualization

We discuss two implementation details for the visualization proposed in this chapter. We first discuss how to create a system that provides a real-time look at the search progress. Then, we discuss how to enforce multiple consistency algorithms during search, as was required in our case study of Section 3.3.

3.4.1 Real-Time Feedback

Previous approaches operate by storing a trace of the program to a text file, allowing a post-mortem examination of the search process. We produce our visualization of the BpD (Section 3.2) in *real time* in order to provide, to the user, an instantaneous feedback of how search is operating.

We implement this visualization in Stampede, the CSP solver developed in the Constraint Systems Laboratory. The framework in Stampede operates by passing JavaScript Object Notation (JSON) messages from the solver (i.e., server) to the web-interface (i.e., client).[3] Messages are cached and sent on a time schedule to avoid the overhead of sending many small messages. More specifically, messages are sent every 3 milliseconds. The client receives these messages and updates the visualization based on the messages received.

Stampede creates a WebSocket server to pass the JSON messages to the web client. The change in the graph from the previous message is sent to the client to reduce the size of the messages sent. Two types of messages are sent from the solver to the client, corresponding to the two visualizations:

1. BpD: The depths where backtracks have occurred since the last message and the additional number of backtracks at those depths.
2. CpD: The depths where each consistency algorithm was enforced since the last message and the result of its enforcement at those depths.

Once a message is received, the interface parses the message and updates the chart, applying the difference to what is currently rendered.

---

[3] Anthony Schneider designed the communication framework for passing messages from Stampede to the visualization. Denis Komissarov designed the generic framework for adding visualizations in JavaScript.

For post-mortem examination of the chart (i.e., examining the chart as it appears at the end of search), no messages are sent to the client. Instead, when the solver terminates, a JSON string representing the complete chart is stored. The web interface is enhanced to also take as input JSON strings.

If a user wants to see the evolution of search, they must watch search as it progresses in real time. However, for problems where search takes a large amount of time, the user is required to wait in real time. Further, the user lacks advanced controls, such as pause and rewind. Howell *et al.* [2018b] extended these visualizations in WORMHOLE to be able to reconstruct the visualization after search terminates, which allows the user to carefully examine the visualization as search progresses. Similar to CP-VIZ [Simonis *et al.*, 2010], WORMHOLE is solver agnostic and can visualize any solver that reports the corresponding JSON messages.

3.4.2 Running Multiple Consistencies

Most consistency algorithms can be viewed as being 'event based' in that a given constraint removes a value in the domain of a variable in its scope requiring some other constraints to be considered. The arc-consistency algorithm AC5 is one of an example of an event-based consistency algorithm [Hentenryck *et al.*, 1992].

Vion *et al.* [2011] fit higher-level consistencies into the same event-based framework by defining a number of abstract constraints depending on how a specific higher-level consistency operates. Each abstract constraint represents a specific combination of element on which the higher-level consistency should operate (e.g., an element in the queue). When this abstract constraint is considered, it executes the higher-level consistency algorithm on that specific combination of elements. In this framework, some of the constraints represent low-level consistency (i.e., AC), others represent high-level consistency. The technique relies on sorting the constraints to determine the proper order of enforcement. These abstract constraints are not able to enforcing consistency in a ‘when’ strategy (i.e., enforce the consistency during certain parts of the search space), but does allow for easily enforcing a ‘where’ strategy (i.e., enforcing the consistency on a part of the problem).

We propose an alternative approach for enforcing multiple consistencies. Our approach does not involve re-framing the consistency algorithms as events. Rather, it ‘informs’ each consistency algorithm about the changes that have occurred in the problem since the consistency was last enforced.[4]

Algorithm 1 shows the steps used when a consistency algorithm, *consistency*, is to be enforced at some *depth* of the search tree. The algorithm uses the following methods:

```
Algorithm 1: RunConsistency($consistency$,$depth$)
Input: $consistency$: A consistency algorithm; $depth$: Depth of the search tree
if $consistency \notin ranConsistencies[depth]$ then SaveState($consistency$)
$ranConsistencies[depth] \leftarrow ranConsistencies[depth] \cup \{consistency\}$
ViewReductions($consistency$)
Run($consistency$)
```

- SaveState(*consistency*) tells the consistency algorithm *consistency* that it is at a new level in search, useful for saving its state of the CSP (e.g., save the supports).

- ViewReductions(*consistency*) passes all of the changes to the problem (i.e., value and tuple deletions) since the last time the consistency algorithm *consistency* was executed. The idea is that the consistency algorithm uses this information

---

[4] This framework was initially designed in collaboration with Nathan Stender. It was later refined and made more efficient in collaboration with Anthony Schneider.

to re-queue any changes for the next time it runs. We discuss below how these changes are stored.

- RUN(*consistency*) executes the consistency algorithm *consistency*.

In Algorithm 1, *ranConsistencies* is a global vector of size n that stores, for each search depth, the set of consistency algorithms executed at that depth. Initially all sets in the vector are empty. Line 1 determines whether or not the consistency was previously executed at this depth by checking *ranConsistencies*, and calls SAVESTATE on the consistency in case it has not been executed before. Line 3 calls VIEWREDUCTIONS on the consistency to alert it of any changes that have occurred since it was last executed.

Every change to the problem (e.g., removing a variable-value pair or a relation-tuple pair) is stored as a *reduction*.[5] Every variable (table constraint) is associated with an ordered list of value (tuple) reductions that were deleted for that variable (table constraint). Every consistency algorithm stores a pointer to every list of reductions, pointing to the latest reduction processed when the algorithm was last enforced. The method VIEWREDUCTIONS retrieves all the reductions listed after the consistency's current pointer then updates this pointer to point to the end of the list. Note that the reductions retrieved may have occurred at any depth of search deeper than when the consistency was last enforced. If a consistency algorithm caused a reduction, it does not need to view these changes as it caused it, thus its pointer is automatically updated to the end of the list. The typical use of the VIEWREDUCTIONS method is to assist the consistency algorithm in what elements of change to re-queue.

When search undoes an assignment (i.e., when a node visit fails), UNDOASSIGNMENT (Algorithm 2) 'tells' the consistency algorithms executed at this search depth to

---

[5] Storing filtered values as reductions was first proposed by Prosser for the Forward-Checking algorithm [Prosser, 1993].

restore the state of the problem. The algorithm UndoAssignment uses the method

Algorithm 2: UndoAssignment(*depth*)

```
Input: depth: The search depth
foreach $consistency \in ranConsistencies[depth]$ do
    RestoreState($consistency$)
$ranConsistencies[depth] \leftarrow \emptyset$
```

RestoreState(*consistency*), which tells the consistency algorithm *consistency* that the assignment is undone. RestoreState corresponds to the undo operation of the SaveState call. RestoreState also restores the pointers of the reductions for every variable and table constraint to the previous level. The restoration of the pointers is accomplished by using a reversible-set data-structure [Demeulenaere *et al.*, 2016].

Summary

In this chapter, we introduced a new approach for visualizing the progression of search by summarizing the number of backtracks and the number of calls to consistency at each depth in the search tree. We also discussed two design mechanisms for implementing this visualization.

Chapter 4

A Reactive Strategy for High-Level Consistency During Search

Enforcing a higher-Level Consistency (HLC) can be between 2–40 times slower than enforcing GAC algorithms. Thus, enforcing HLC needs to reduce the number of node visits by this same amount for enforcing HLC to yield CPU time improvements. Alternatively, instead of enforcing HLC at every step in search we can selectively enforce it.

Our motivation comes from noticing that using the variable-ordering heuristic dom/wdeg [Boussemart *et al.*, 2004] with GAC algorithms is able to solve many problems with little search. In these situations, we want to exploit the easiness of the problem by letting GAC solve the problem and not enforcing HLC.

In this chapter, we present PrePeak$^+$ as a reactive strategy that operates on the two dimensions 'when' and 'how much.' In particular, (*a*) we introduce a triggering strategy, PrePeak, that tracks search performance and triggers HLC when search starts thrashing (i.e., when), and (*b*) choose to enforce HLC on a fraction of the (ordered) propagation queue and within a bounded time duration (i.e., how much). We validate our approach on benchmark problems using Partition-One Arc-Consistency as an HLC. However, our strategy is generic and can be used with other higher-level consistency algorithms, as we show in future chapters.

4.1 When HLC: A Trigger-Based Strategy

We first introduce our HLC-triggering strategy, PrePeak. Then we discuss using geometric laws to allow PrePeak to react to the effectiveness of enforcing HLC.

4.1.1 PrePeak

The idea behind our reactive strategy is to monitor the 'progress' of search while maintaining some consistency property, such as GAC, in a d-way branching backtrack search. When search starts thrashing, we trigger some high-level consistency (HLC), such as POAC, and keep enforcing it as long as it is beneficial. In order to determine that thrashing has reached a dangerous level, we propose to track the number of backtracks at each depth (or level) of the search tree. We advocate using the number of backtracks as a better indication of thrashing than the number of constraint checks (e.g., [Epstein *et al.*, 2005]) or the number of nodes visited because the former depends on the number of constraints that apply to a variable and the latter depends on the variable's domain size. To this end, we store the number of times each level of the search tree was backtracked to in a vector $btcounts[\cdot]$ indexed by the corresponding level. The size of the vector is $n+1$ where n is the number of variables in the problem. When an entry in this vector reaches some threshold value θ, we set $peak_d$, identified as the 'peak' depth of thrashing, to the search depth corresponding to that entry. When search backtracks to a shallower depth than $peak_d$, we enforce HLC as long as HLC is effective, then we revert to enforcing GAC after resetting to 0 all the counts in *btcounts*$[\cdot]$. We call this approach PrePeak because (a) it is based on identifying the peak depth to which search backtracks and (b) HLC is enforced up to this depth. Our goal is to 'hit hard' the future subproblem with HLC and reduce its size before the search reaches the peak depth again.

We present PrePeak as simple modifications of the functions Unlabel (Algorithm 3) and Label (Algorithm 4) of Prosser's 'classical' backtrack search algorithm [1993]. These modifications are obtained by adding the lines highlighted in the pseudocode. Below, we discuss only the lines corresponding to our modifications. Further, we declare *btcounts*$[\cdot]$ and $peak_d$ as global variables to the search procedure. We initialize all the entries of *btcounts*$[\cdot]$ to 0 and set $peak_d$ to 0 indicating that there is no active peak.

```
Algorithm 3: Unlabel($i$,$consistent$) unlabels variable $x_i$
   Input: $i$: depth of failed variable; $consistent$: state of current path
   Output: depth of current variable
1  Restore domains of current and future variables
2  $h \leftarrow i - 1$
3  $dom(x_h) \leftarrow dom(x_h) \setminus \{\text{AssignedValue}(x_h)\}$
4  $consistent \leftarrow dom(x_h) \neq \emptyset$
5  $btcounts[h] \leftarrow btcounts[h] + 1$
6  if $btcounts[h] = \theta$ then $peak_d \leftarrow h$
7  return $h$
```

In Line 5 of Unlabel (Algorithm 3), we increment the value of *btcounts*$[h]$ where h is the depth to which we backtrack. If *btcounts*$[h]$ reaches the threshold value θ, we set $peak_d$ to h to reduce the chance of thrashing at i (Line 6). We discuss the selection of the initial value of θ in Section 4.1.3.

It is in the function Label (Algorithm 4) that we must decide whether or not to enforce HLC. At every assignment of the current variable x_i, we first enforce GAC (Line 6). At Line 7, if we find that a peak was identified ($peak_d > 0$) and the

Algorithm 4: LABEL(i,*consistent*) instantiates variable x_i

```
   Input: $i$: depth of current variable; $consistent$: state of current path
   Output: depth of current variable
 1 $consistent \leftarrow false$
 2 $HLCenforced \leftarrow false$
 3 $HLCfiltered \leftarrow false$
 4 foreach $v_i \in dom(x_i)$ while not $consistent$ do
 5     $x_i \leftarrow v_i$
 6     $consistent \leftarrow \mathrm{GAC}(\mathcal{P})$
 7     if $consistent$ and $peak_d > 0$ and $i \leq peak_d$ then
 8         $(consistent, filtered) \leftarrow \mathrm{HLC}(\mathcal{P})$
 9         $HLCenforced \leftarrow true$
10         $HLCfiltered \leftarrow HLCfiltered$ or $filtered$
11     if $not\ consistent$ then $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
12 if $HLCenforced$ then
13     if not $consistent$ then $\theta \leftarrow r_w \cdot \theta$
14     else
15         $\forall z\ btcounts[z] \leftarrow 0$
16         $peak_d \leftarrow 0$
17         if $HLCfiltered$ then $\theta \leftarrow r_f \cdot \theta$
18         else $\theta \leftarrow r_n \cdot \theta$
19 if $consistent$ then return $i + 1$ else return $i$
```

current depth is shallower than the peak's depth ($i \leq peak_d$), we enforce HLC on the future subproblem recording the outcome of this call, for the given assignment, using the Boolean variables *consistent* and *filtered* (Line 8), where *consistent* indicates the consistency of the current path and *filtered* indicates whether or not HLC yielded any filtering. The Boolean variables *HLCenforced* and *HLCfiltered* indicate, for any tested assignment for the current variable, whether or not HLC was enforced (Line 9) and yielded filtering (Line 10), respectively. Note, once HLC is triggered, we enforce it for all the tested values for the current variable x_i.

We claim that, whenever we trigger HLC, it is timely to revise and update the triggering threshold, θ, given the recorded outcome of HLC. We distinguish three regimes:

1. *Wipeout*: HLC effectively depletes the domain of x_i by yielding a wipeout at every instantiation. It forces search to backtrack.
2. *Filtering*: HLC yields some filtering, but finds a consistent assignment for x_i and allows search to proceed to the next level.
3. *Neither*: HLC does not yield any filtering at all (beyond what GAC may have filtered). Search proceeds to the next level with a consistent instantiation for x_i.

We update the threshold value θ by multiplying its current value by a factor of r_w (Line 13), r_f (Line 17), or r_n (Line 18), for each of the above regimes, respectively, as we argue below. We discuss these factors in Section 4.1.2.

The first regime (i.e., wipeout) 'reinforces' our belief in the usefulness of HLC and entices us to continue to enforce HLC as we backtrack by one or more levels. To this end, we do not reset the values of $peak_d$ or $btcounts[\cdot]$. In the remaining two regimes, we are reserved about the usefulness of HLC and prevent it from triggering again too soon. Thus, we reset the values of both $btcounts[\cdot]$ and $peak_d$ (Lines 15 and 16). As a result, subsequent calls to the function Label do not enforce HLC until a new peak is detected.

4.1.2 Update Strategies for θ

The three identified regimes allow us to 'plug in' arbitrary strategies for updating θ, thus providing an opportunity to adjust PrePeak's reactivity to the relative cost of the consistency properties enforced. We propose to use geometric laws similar to the one used for cutoff-value update in restarting randomized search [Walsh, 1999] $\theta \leftarrow r \cdot \theta$ with different values of the common ratio r for each of the regimes.

1. Wipeout: We use $r_w = 1.2^{-1}$ (Line 13).[1]
2. Filtering: We use $r_f = 1.2^2$ (Line 17).
3. Neither: We use $r_n = 1.2^3$ (Line 18).

The above strategies allow PrePeak to adapt to the instance at hand by updating θ based on HLC's pruning effectiveness. Indeed, when it yields a domain wipeout (first regime), HLC is effectively reducing thrashing and its frequency is increased. Otherwise, the update strategies decrease HLC's triggering frequency more aggressively when HLC yields no filtering (third regime) than when it does (second regime).

4.1.3 Initializing the threshold θ

Our reactive strategy enforces GAC until search starts thrashing, triggering HLC when backtracking 'reaches' the value of the threshold θ. If we choose too small an initial value for θ, HLC may trigger while GAC is still effective, which adds to the CPU cost.[2] If we choose too large a value, GAC may have run for too long in a barren subtree.

In PrePeak, the distribution of the backtracks in the vector *btcounts*$[\cdot]$ varies depending on the problem instance, making the choice of the initial value of θ not straightforward. We investigated an alternative reactive strategy that triggers HLC

---

[1] The value of 1.2 is a commonly used factor (e.g., [Walsh, 1999]) and provides a 'gentle' evolution. Other values tested (e.g., 1.1, 1.4, and 1.6) yielded qualitatively similar results.

[2] We empirically noticed that the three update laws (Section 4.1.2) allow us to recover from starting with smaller values by dynamically adjusting the value of θ to the instance at hand, thus providing some robustness to our approach.

based on the value of $\sum_{l=1}^{n} btcounts[l]$. This study inspired the following initialization of θ for PrePeak: we set θ to be the maximum value of $btcounts[\cdot]$ when $\sum_{l=1}^{n} btcounts[l] = n^2$, thus, setting $\theta \leftarrow \max_{l=1}^{n}(btcounts[l])$. In other words, we identify the first peak and its depth by taking a snapshot of the backtrack profile after search executes n^2 backtracks. We tested different values, such as various powers of n, various factors of n, the sum of domain sizes, and the ratio of the CPU times for enforcing GAC and HLC computed in a pre-processing step. We empirically found that values that are quadratic in the number of variables (e.g., n^2 and sum of domain sizes) perform best, thus, we select n^2.

4.2 How Much HLC: Monitoring Propagation

We propose using two mechanisms to control the early termination of HLC, namely, the size of the propagation queue and the time bound for running HLC:

1. While ordering the elements of the propagation queue of the HLC algorithm based on the activity of a variable or constraint (e.g., dom/wdeg [Boussemart *et al.*, 2004]), we allow only a fraction of the propagation queue to be processed.
2. We impose a bound on the duration of any call to HLC.

Let q be the number of elements in the propagation queue each time we trigger HLC. We terminate HLC as soon as either of the following two criteria is met:

1. $\frac{q}{2}$ elements of the propagation queue are processed, or
2. HLC has consumed a total CPU time $\frac{q}{2} \cdot \textsc{Time}(\text{GAC})$ where $\textsc{Time}(\text{GAC})$ is the time spent on the last call to GAC prior to HLC (Line 6 of Algorithm 4).

Our approach is inspired from Balafrej *et al.* [2014], who noticed that POAC is too costly to be used on its own. They advocated to (a) order the variables in the propagation queue by the dom/wdeg ordering heuristic and (b) terminate POAC when the amount of filtering by POAC significantly drops. They proposed an adaptive strategy APOAC, based on a "10% learning, 90% exploitation"-learning strategy, which assumes that POAC is enforced at every step during search. PrePeak cannot accommodate such a learning process because HLC is enforced only reactively.

Other mechanisms to monitor propagation may exist. For example, we can watch propagation during a given window of the propagation queue while sliding this observation window as long as filtering is 'active.' Alternatively, we can consider a sliding window of time. We tested combinations of such criteria. While the results were positive in general, they were unstable across benchmarks. As a lesson, we conclude that a fixed amount for each mechanism (i.e., queue and time) is simpler to implement, more stable, and as effective.

4.3 Other Reactive Triggering Strategies

Reactive triggering is a general strategy of which PrePeak is one instance. We present alternative instances for comparison.

4.3.1 BTWatch

We introduce another reactive triggering strategy that we call BTWatch. The idea of BTWatch is to maintain a *single* backtrack counter throughout the search process that we compare to the threshold value θ in the same manner as in PrePeak. As a result, BTWatch may trigger at a search depth shallower or deeper than the peak's depth. Thus, in BTWatch, we determine that search is thrashing by watching the backtrack counts during search, while in PrePeak, we do so by tracking the peak in the backtrack profile. Before discussing BTWatch, we first describe a hybrid, PP-BTWatch. Empirically, all three strategies are statistically equivalent, but they improve our understanding of reactive strategies.

PP-BTWatch aims at controlling the depth at which HLC is triggered in BTWatch, the rationale being that we need to 'pound on the difficulty' right before it arises. To this end, in PP-BTWatch, we compute the backtrack counter of BTWatch as $\sum_{l=1}^{n} btcounts[l]$ and the trigger depth of PrePeak as $\arg\max_{l=1}^{n}(btcounts[l])$. In practice, PP-BTWatch uses the functions Unlabel (Algorithm 3) and Label (Algorithm 4) of Section 4.1.1 by simply changing Line 6 of Unlabel (Algorithm 3) as follows:

$$\textbf{if } \sum_{l=1}^{n} btcounts[l] = \theta \textbf{ then } peak_d \leftarrow \arg\max_{l=1}^{n}(btcounts[l])$$

BTWatch triggers HLC every θ backtracks at any depth and regardless of $peak_d$. BTWatch uses the same function Unlabel as PP-BTWatch (i.e., with the modified Line 6 in Algorithm 3). As for the function Label, BTWatch removes the test $i \leq peak_d$ in Line 7 of Algorithm 4 so that it triggers HLC as soon as the threshold θ is met. Note that, for BTWatch, we could dispose of $btcounts[\cdot]$ and use a simple integer as a backtrack counter.

We designed and investigated PrePeak and BTWatch in parallel. Comparing their behavior improved our insight into reactive triggering and allowed us to blend the two strategies. For example, PP-BTWatch implements the use of $peak_d$ in BTWatch. Conversely, PrePeak borrows the initialization mechanism of θ of BTWatch. In our experiments, all three strategies yielded statistically equivalent results. We believe that more triggering strategies remain to be investigated.

4.3.2 Scheduled Enforcement of HLC

The most basic We introduce three strategies that decide when to enforce HLC based on a schedule rather than using feedback from the progression of search.

Random: We introduce a random strategy for triggering HLC as a baseline comparison. In our experiments, we randomly trigger 1% or 10% of the time.

Time: Recent work in the SAT community has been applying higher-level consistency to SAT problems both at pre-processing [Davis and Putnam, 1960; Rish and Dechter, 2000; Subbarayan and Pradhan, 2005; Eén and Biere, 2005], and inprocessing (i.e., lookahead) [Järvisalo *et al.*, 2012]. The higher-level consistency enforced is typically a form of variable elimination or bucket elimination in CP terminology [Seidel, 1981; Dechter and Pearl, 1988]. The inprocessing done by SAT is conducted by interleaving search and enforcing high-level consistency. Wotzlaw *et al.* [2013] advocate reserving 10% of the CPU time for inprocessing versus search. As a result, the more time is spent on search, the more 'inprocessing' is allowed.

The TIME strategy mimics that of Wotzlaw *et al.* [2013], which monitors the cumulative time spent enforcing HLC, CUMULATIVETIME(HLC), and the cumulative time spent enforcing GAC, CUMULATIVETIME(GAC). When $\frac{\textsc{CumulativeTime}(HLC)}{\textsc{CumulativeTime}(GAC)} < x$, for some value x, we will allow HLC to be enforced.

TimeRatio: The TIMERATIO strategy is similar to TIME, except that it removes the parameter x (i.e., 10%). Instead, at pre-processing it determines the amount of time for enforcing GAC, $\textsc{RunningTime}(GAC)$, followed by the enforcement of HLC, $\textsc{RunningTime}(HLC)$. Using the ratio $\frac{\textsc{RunningTime}(GAC)}{\textsc{RunningTime}(HLC)} = x$, TIMERATIO trigger HLC after every x calls to GAC.

4.4 Empirical Evaluation on POAC

In this section, we evaluate the effectiveness of our strategy. To this end, we consider the problem of finding a single solution to a CSP using backtrack search, the dom/wdeg variable ordering heuristic [Boussemart *et al.*, 2004], and real-full lookahead [Haralick and Elliott, 1980].

We first discuss our experimental setup. We then validate our approach in five directions:

1. We demonstrate that our strategy is better than triggering randomly, Section 4.4.3.
2. We show that PrePeak$^+$ performs better than either of its components, Section 4.4.4
3. Compare PrePeak$^+$ against GAC and APOAC using two different *dynamic* variable-ordering heuristics: dom/deg [Bessière and Régin, 1996] and dom/wdeg [Boussemart *et al.*, 2004], Section 4.4.5
4. We introduce a visualization of the search process to provide a graphical interpretation of the good performance of our approach, Section 4.4.6.
5. Comparison to a multi-armed bandit technique, Section 4.4.7.

4.4.1 Experimental Setup

We set up our experiments as follows. We use STR2+ [Lecoutre, 2011] as the GAC algorithm for lookahead (Line 6 in Algorithm 4) because STR2+ empirically outperforms GAC2001 on non-binary problems.

We choose POAC for the higher-consistency property and enforce it using the POAC-1 algorithm [Balafrej *et al.*, 2014], where we exclude variables with singleton domains from the singleton tests. We use the benchmark problems available from Lecoutre's website.[3] We test all available binary and non-binary CSPs, including Boolean, patterned, random, quasi-random, academic, and real-world benchmarks. We include all benchmarks with at least one instance with a primal graph of density less than 50%.[4] Indeed, on high density networks, the impact of an instantiation on a future variable is immediately propagated by GAC while HLC typically yields no further filtering but costs predictable data-setup overhead. This selection results in a total of 138 benchmarks (57 non-binary and 81 binary) consisting of 4,077 instances (1,716 non-binary and 2,361 binary). The selected benchmarks have a mixture of instances with densities $\geq 50\%$ and $< 50\%$, however, only 137 instances of the 4,077 instances included have densities $\geq 50\%$. We setup our reactive strategies to first compute the density of an instance. If the density is $\geq 50\%$, we enforce GAC. Otherwise, we execute the reactive strategy. Our results include this computation time. We use a time limit of 60 minutes per instance and 8GB of memory.

We denote PrePeak$^+$ the combination of our when strategy (PrePeak, Section 4.1) and our how-much strategy (Section 4.2).

In the tables that summarize our results, we report for each algorithm, where applicable:

- The number of instances solved in a given benchmark (#solved)
- The number of node visits averaged over the instances completed by all algorithms (avg. NV)

---

[3] `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

[4] Table E.1 in Appendix E list the selected benchmarks.

- The sum of the CPU time in seconds of the run time of an algorithm for all the instances in a benchmark completed by any of the compared algorithms ($\sum$CPU). When an algorithm does not terminate within the allocated time, we add 3,600 seconds to the CPU time and indicate with a '>' sign that the time reported is a lower bound.

- The average number of calls to POAC over the instances completed by all algorithms (#CallsPOAC). GAC does not call POAC, thus the number of calls to POAC is reported as N/A.

- Finally, we highlight, with a boldface, the best value in a given row.

4.4.2 Comparing with BTWatch

We compare the performance of PrePeak$^+$ to that of the BTWatch and PP-BTWatch strategies of Section 4.3.1. We again denote BTWatch$^+$ and PP-BTWatch$^+$ the combination of BTWatch and PP-BTWatch and our how-much strategy (Section 4.2). Table 4.1 gives the overall performance on all the benchmarks evaluated. All three strategies are statistically equivalent. However, PrePeak$^+$ solves the largest number of instances in the smallest CPU time. Thus, we discuss only PrePeak$^+$ as it offers the best empirical performance. Table F.1 of Appendix F gives the data for each benchmark.

Table 4.1: The overall performance of the BTWatch strategies of Section 4.3.1

	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
#solved	**2,284**	2,278	2,279
ΣCPU [sec]	>**338,775.9**	>341,106.8	>343,723.7
avg. NV	412,568.6	**327,001.4**	364,717.8
#CallsPOAC	723.8	486.6	645.4
#Instances: total 4,077; solved by all 2,269; solved by one 2,291			

4.4.3 Triggering Cannot be Scheduled

We compare the performance of PrePeak$^+$ to that of the three 'scheduled' strategies discussed in Section 4.3.2, namely, Random, TimeRatio, and Time, which we combine with our how-much strategy (Section 4.2). For both Random and Time, we try two parameters, using 10% and 1% for the percentage of effort to spent on HLC. Table 4.2 gives the overall performance on all the benchmarks evaluated. PrePeak$^+$ solves the largest number of instances in the least amount of time. All of the Random, TimeRatio, and Time strategies are worse than GAC. Random-10%, TimeRatio, and Time-10% have a large number of calls to POAC compared to PrePeak$^+$. One may think that the large number of calls to POAC is the cause of the performance loss. Thus, we evaluated the Random-1% and Time-1% strategies, which lowered the calls to POAC to be on the same order of magnitude than PrePeak$^+$, they continued to be worse than GAC. Looking at the average node visits, Random-1% and Time-1% visited the same order of node visits as GAC, *despite enforcing more POAC than* PrePeak$^+$. This results shows that PrePeak$^+$ is correctly targeting the areas to enforce POAC to reduce the search space. Overall, Random, TimeRatio, and Time yield poor results because they are agnostic to 'where,' in the search space, an HLC is needed. Further, they are unable to react to the effectiveness of HLC (i.e., amount of pruning obtained by the HLC).

Table 4.2: The overall performance of the other strategies of Section 4.3.2

			Random			**Time**	
Algorithm	**GAC**	**PrePeak$^+$**	**1%**	**10%**	**TimeRatio**	**1%**	**10%**
#solved	2,278	2,286	2,271	2,272	2,270	2,276	2,275
$\sum$CPU [sec]	>365,233.4	>349,578.1	>391,457.7	>419,706.0	>394,812.8	>372,481.8	>393,591.7
avg. NV	473,637.8	324,265.7	447,399.8	191,179.6	276,524.4	468,868.4	385,157.7
#CallsPOAC	-	283.6	783.8	4,324.9	3,632.9	501.7	1,622.9
#Instances 4,077 total, 2,248 by all, 2,296 by at least one							

4.4.4 Putting together 'When' and 'How Much'

In this experiment, we use the dom/wdeg variable-ordering heuristic. Table 4.3 shows the contributions of the two aspects 'when' (PrePeak, Section 4.1) and 'how much' (interrupting propagation, Section 4.2) to the good performance of PrePeak$^+$. PrePeak$^+$ solves more instances than either component taken individually, which shows the importance of combining the two orthogonal dimensions. The number of calls to POAC in PrePeak$^+$ is mostly controlled by the triggering strategy (i.e., 'when'), which by itself is more expensive than PrePeak$^+$ because POAC runs until a fixpoint. The right-most column enforces POAC with early termination (Section 4.2) at *every* node, yielding the smallest number of nodes visited but the largest CPU time. 'When' and 'how much' complete difference instances: only 2,131 instances are completed by both. Combining 'when' and 'how much' in PrePeak$^+$ allows it to solve instances not solved by both.

Table 4.3: PrePeak$^+$ versus 'when,' 'how much'

Algorithm	**PrePeak$^+$**	**When**	**How Much**
#solved	**2,286**	2,239	2,171
∑CPU [sec]	>**356,778.1**	>610,958.6	>915,738.6
avg. NV	568,072.7	123,224.9	**10,925.7**
#CallsPOAC	2,477.5	1,128.1	5,019.4
#Instances: 4,077 total; 2,131 by all; 2,298 by at least one			

4.4.5 PrePeak$^+$ versus GAC and APOAC

Table 4.4 compares the performance of GAC, APOAC, and PrePeak$^+$ under the dom/deg ordering heuristic.[5] PrePeak$^+$ solves the most instances and is the fastest

---

[5] Although dom/wdeg is generally more effective than dom/deg, the decisions made by dom/wdeg are considered too unstable to objectively allow comparing algorithms' performance. Researchers studying the performance of HLC during search typically use dom/deg in their experiments [Balafrej *et al.*, 2015; Paparrizou and Stergiou, 2016; Paparrizou and Stergiou, 2017].

Table 4.4: GAC, APOAC, and PrePeak$^+$ on dom/deg

Algorithm	**GAC**	**APOAC**	**PrePeak$^+$**
#solved	2,036	2,058	**2,173**
∑CPU [sec]	>1,044,380.1	>1,042,622.9	>**455,189.2**
avg. NV	1,138,447.6	**90,047.4**	324,020.2
#CallsPOAC	-	30,911.5	686.1
#Instances: 4,077 total; 1,891 by all; 2,205 by at least one			

algorithm. Predictably, in terms of average nodes visited, APOAC explores the fewest and PrePeak$^+$ is closer to APOAC than to GAC despite the relatively few calls to POAC (686.1). We conclude that PrePeak$^+$ triggers HLC at the right place and in the right amount, thus validating our approach.

Figure 4.1 shows the cumulative number of instances completed by GAC, APOAC, and PrePeak$^+$ (on dom/deg) as time increases. Comparing GAC and APOAC, we see that GAC dominates APOAC on instances solved within 1,600 seconds, while APOAC dominates GAC after this point. This behavior motivates the need for HLC on difficult instances and illustrates its overhead on easier instances. By selectively triggering HLC, our strategy, PrePeak$^+$, dominates both GAC and APOAC.

Table 4.5 repeats the same experiment under dom/wdeg. The results are similar to those in Table 4.4: PrePeak$^+$ outperforms GAC and APOAC in terms of both number of instances solved and CPU time. Note that it would be incorrect to conclude that the CPU time of APOAC deteriorates from Table 4.4 to Table 4.5 because this

Table 4.5: GAC, APOAC, and PrePeak$^+$ on dom/wdeg

Algorithm	**GAC**	**APOAC**	**PrePeak$^+$**
#solved	2,279	2,138	**2,286**
∑CPU [sec]	>372,433.4	>1,095,125.8	>**356,778.1**
avg. NV	480,897.9	**23,472.9**	319,453.4
#CallsPOAC	-	9,924.4	288.6
#Instances: 4,077 total; 2,122 by all; 2,298 by at least one			

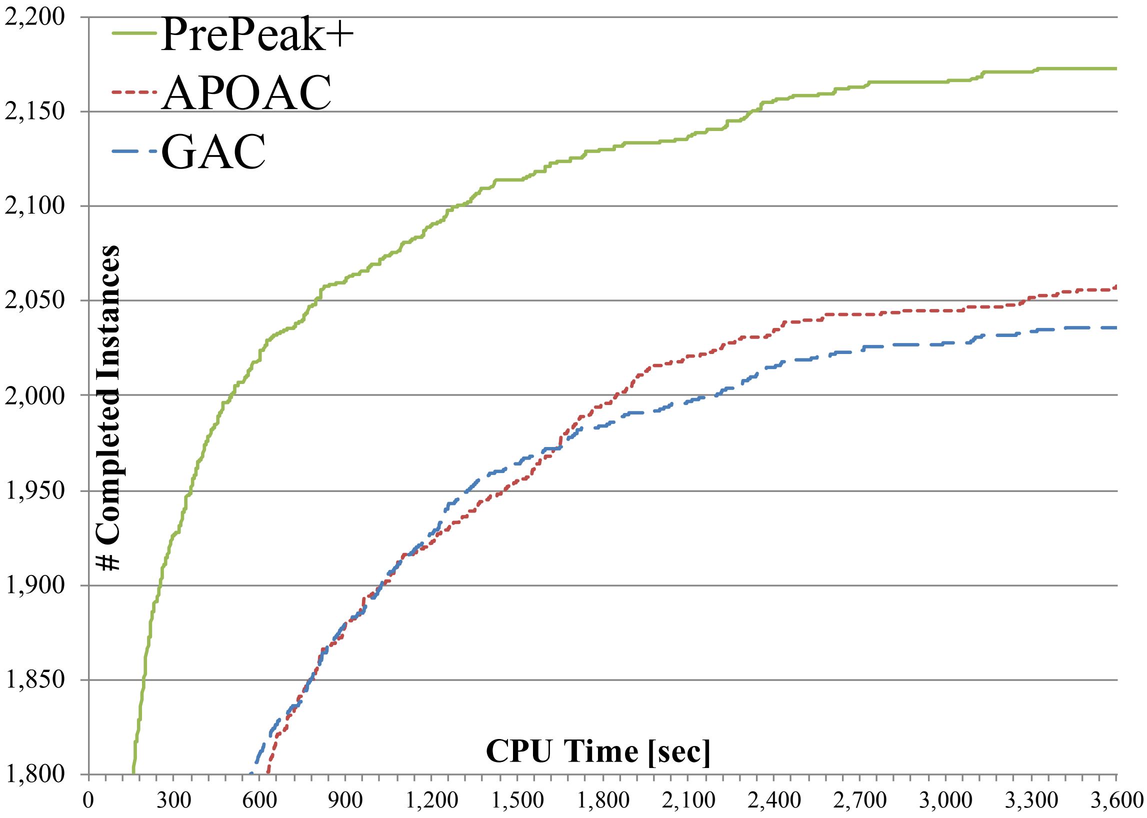


Figure 4.1: Cumulative instances completed by CPU time on dom/deg

measurement accounts for the number of instances completed in each experiment, which is different (i.e., 2,205 in Table 4.4 and 2,298 in Table 4.5).

Figure 4.2 shows the cumulative number of instances completed by GAC, APOAC, and PrePeak$^+$ (dom/wdeg) as CPU time increases. APOAC is clearly dominated by both GAC and PrePeak$^+$. For instances easily solved by GAC (i.e., solved in less than 300 seconds), PrePeak$^+$ has few calls to POAC because GAC is not thrashing. For the remaining harder instances, PrePeak$^+$ dominates GAC. PrePeak$^+$ remains competitive under dom/wdeg, which is known to dwarf the benefits of HLC.

Table 4.6 provides a finer examination of the results with dom/wdeg for a range of representative benchmarks, showing the number of instances in each benchmark in parentheses.

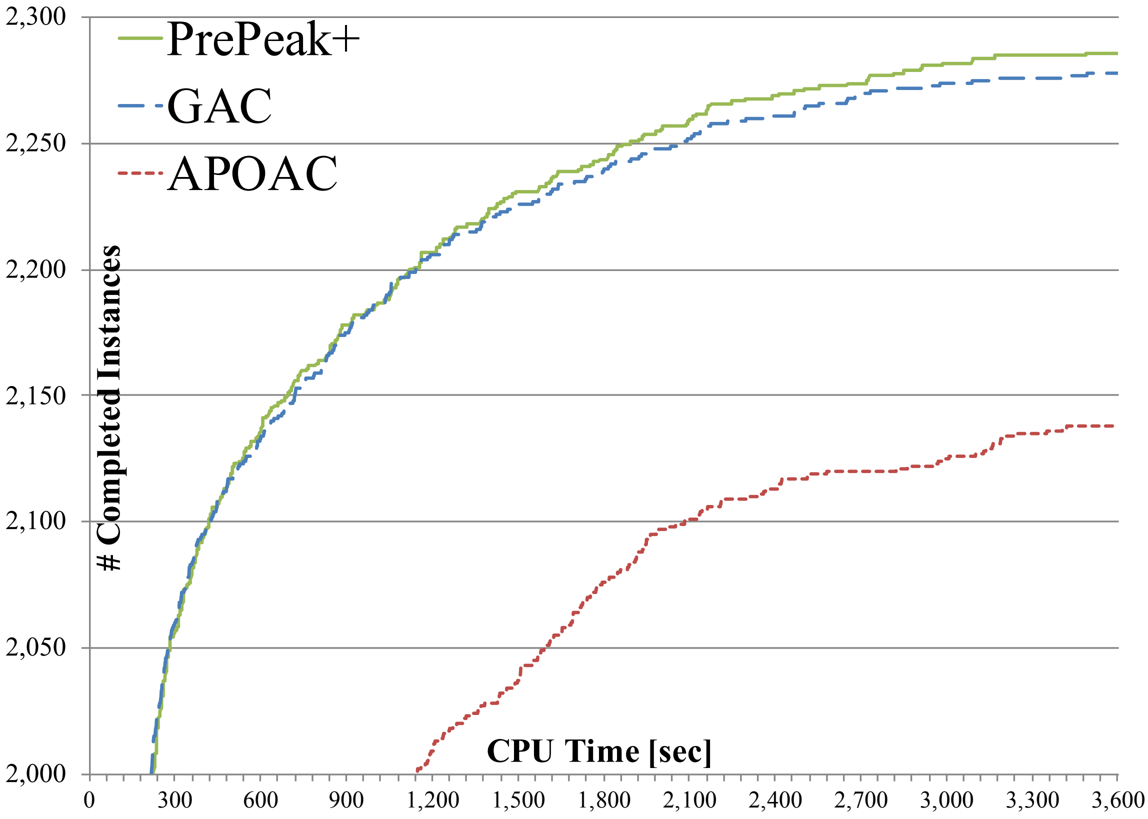


Figure 4.2: Cumulative instances completed by CPU time on dom/wdeg

Rows 1–3 show benchmarks where PrePeak$^+$ significantly outperforms all others both in CPU time and the number of solved instances. For all remaining benchmarks, PrePeak$^+$ solves as many instances as the best algorithm.

APOAC solves more instances than GAC in rows 4 and 5, showing that HLC is required for these benchmarks. PrePeak$^+$ solves the same number of instances as APOAC, in faster CPU time, by selectively enforcing HLC. These benchmarks confirm the ability of our approach to mimic APOAC's performance when APOAC is needed.

In row 6 (QCP-15), GAC and APOAC are roughly equivalent, yet PrePeak$^+$ outperforms both in CPU time. For rows 7 and 8, GAC solves more instances than APOAC, however, PrePeak$^+$'s few calls to POAC allow it to slightly improve on the

Table 4.6: Representative benchmarks using dom/wdeg (time in [sec])

	Benchmark		**GAC**	**APOAC**	**PrePeak$^+$**
1	QCP-20	# solved	4	4	**5**
	(15)	∑CPU	>5,328.7	>4,861.0	**2,762.9**
2	nengfa	# solved	4	4	**5**
	(10)	∑CPU	>3,820.6	>4,235.9	**2,321.0**
3	frb45-21	# solved	7	0	**8**
	(10)	∑CPU	>17,642.0	>28,800.0	**16,239.8**
4	k-insertion	# solved	16	**17**	**17**
	(32)	∑CPU	>3,955.2	3,550.0	**2,903.5**
5	pseudo-ii	# solved	9	**14**	**14**
	(41)	∑CPU	>18,619.8	2,481.9	**2,088.4**
6	QCP-15	# solved	**15**	**15**	**15**
	(15)	∑CPU	1,310.0	1,248.4	**1,213.5**
7	sgb-queen	# solved	**14**	12	**14**
	(50)	∑CPU	5,712.9	>9,969.6	**5,692.0**
8	super-os	# solved	**9**	1	**9**
	taillard5 (30)	∑CPU	11,971.1	>28,924.3	**11,969.8**
9	super-os	# solved	**28**	22	**28**
	taillard-4 (30)	∑CPU	**7,647.2**	>33,042.7	7,675.5
10	geom	# solved	**100**	98	**100**
	(100)	∑CPU	**7,254.3**	>28,365.6	7,372.8
11	TSP-20	# solved	**15**	**15**	**15**
	(15)	∑CPU	**276.5**	1,426.9	298.6

CPU performance of GAC. For rows 9–11, GAC significantly outperforms APOAC both in instance completions and CPU time: HLC is too costly on these benchmarks. However, PrePeak$^+$ is able to adapt to the situation with a CPU time similar to GAC's.

4.4.6 Visualizing Search Performance

For a deeper insight into the behavior of search, we visualize the search execution, using dom/wdeg, on a CSP instance as shown in Figure 4.3, which 'profiles' search with GAC, APOAC, and PrePeak$^+$. In each of the three plots, we report, on the horizontal axis, the depth of the search tree. We plot the number of backtracks at each depth, accumulated throughout search (purple line), with the scale reported on the vertical axis to the *left*. We superimpose the cumulative number of calls to POAC (#Calls POAC) at each depth, with the scale reported on the vertical axis to the *right*. We split the number of calls to POAC into three cases: POAC yields wipeout (green line), POAC yields some filtering (blue line), and POAC yields no filtering at all (red line).

In Figure 4.3, the backtrack curve (purple) shows that APOAC (middle) dramatically reduces the peak value reached by GAC (top) thanks to the large number of calls to POAC. Unfortunately, many of these calls are totally wasted (red curve) or likely of little impact (blue curve): In the middle plot, they compete with the wipeout calls (green curve). PrePeak$^+$ (bottom) makes significantly fewer calls to POAC and those calls are mostly effective (many more calls in green than in blue or red), which establishes that HLC is wisely exploited.

4.4.7 Comparison to Multi-Armed Bandits

Balafrej *et al.* [2015] use Multi-Armed Bandits (MABs) at each search level to choose among a set of consistency algorithms. We compare this MAB technique to PrePeak$^+$.[6] To level the playing field, we enhanced them both with our propagation-monitoring strategy (i.e., ‘how-much HLC,’ Section 4.2).

In our experiments on dom/wdeg (see Table 4.5), the MAB approach solves 2,253 instances in >529,767.9 seconds. It outperforms APOAC but performs worse than GAC. Because each MAB operates at a fixed level in search, using dom/wdeg adversarially affects the effectiveness and stability of a bandit’s learning. Further, the

---

[6] We choose between GAC and POAC although the original paper also uses maxRPC, but it operates on only binary CSPs.

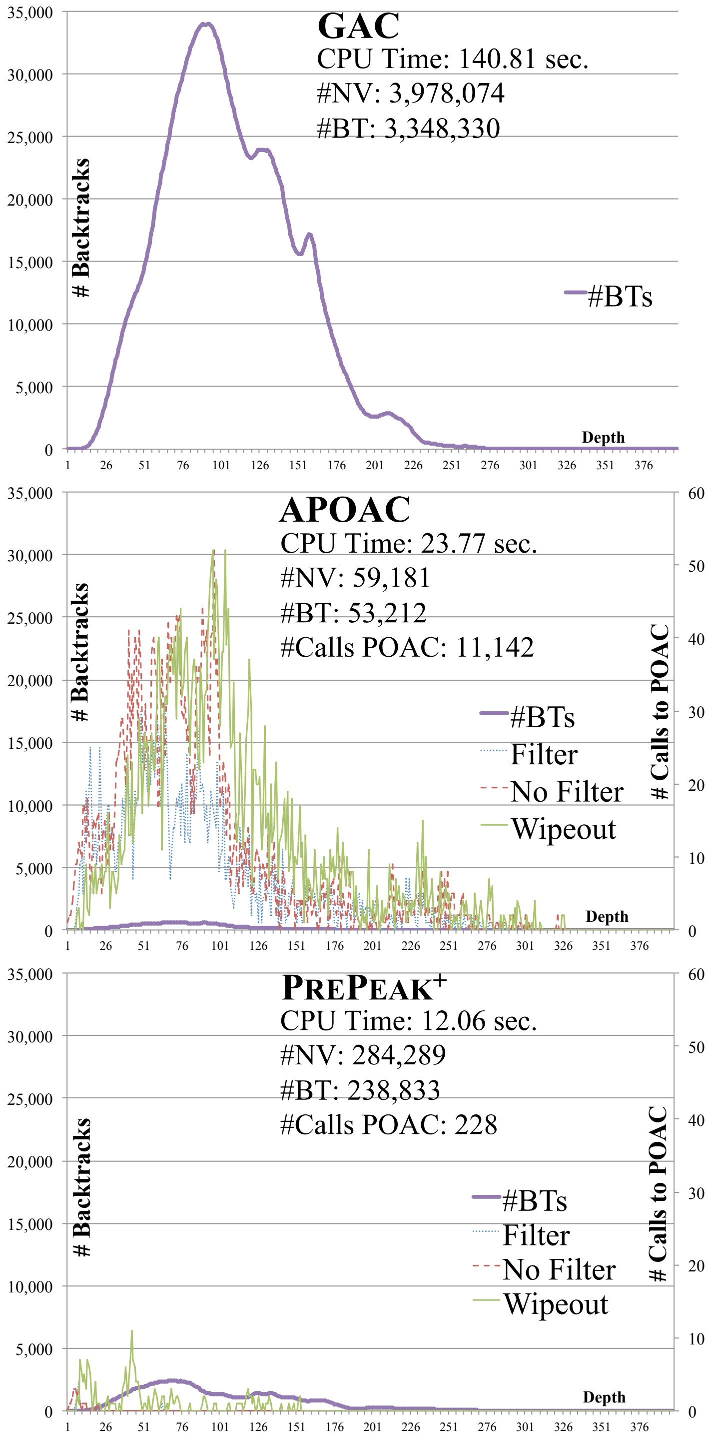


Figure 4.3: Search progress on pseudo-aim-200-1-6-4 using dom/wdeg: GAC (top), APOAC (middle), and PrePeak$^+$ (bottom)

MAB approach assesses the performance of a consistency call by the CPU cost of searching the subtree rooted at the call (regardless of which consistencies are used in the subtree). PrePeak$^+$ largely outperforms the MAB-based strategy for both dom/deg and dom/wdeg because PrePeak$^+$ uses the number of backtracks to assess search progress, which is a more precise measure of the HLC's effectiveness.

Summary

In this chapter we introduce a simple, reactive, trigger-based strategy for advantageously enforcing a higher-level consistency during search, which we call PrePeak$^+$, and empirically validate our approach.

Chapter 5

Restricting Consistency to Cycles

The goal of this chapter is to provide a solution to the question of where to enforce consistency. In particular, we investigate looking at the cycles that appear in a graphical structure of the CSP. The rationale for investigating cycles structures is because a cycle is the basic graphical component as to why arc consistency fails, and thus, are a basic component for the complexity of solving the CSP.

We first introduce the theoretical benefits of utilizing the cycles with singleton consistencies, proving situations where the CSP becomes tractable if it possesses certain structural restrictions. Then, we introduce a technique for localizing cycles to POAC followed by RNIC. Finally we give an empirical evaluation of the approach.

5.1 New Conditions for Tractability

We focus on the *structural* restrictions to a CSP that cause it to be solvable in a backtrack free manner (i.e., becomes tractable). Another line of research is formalizing *language* restrictions (i.e., how constraints are formed) and hybrid restrictions restricting both language and structure [Cohen and Jeavons, 2017]. There is likely a unifying framework, but that is out of the scope of our work.

We start with the most basic form of structural tractability: an acyclic CSP can be solved in a backtrack tree manner after enforcing Directional Arc-Consistency [Dechter and Pearl, 1988; Freuder, 1982].

If the CSP contains exactly one cycle, enforcing SAC on the problem breaks the cycle and it can be solved backtrack free. If there are two cycles and their overlap is on more than one variable, or there are three cycles that have two disjoint overlaps, SAC cannot solve the problem because singleton testing a individual variable cannot break all of the cycles simultaneously. This example motivates our analysis, to determine under what conditions SAC can break all the cycles.

We next discuss some basic terminology for our analysis, then discuss the conditions on binary and then non-binary CSPs.

5.1.1 Terminology

For our analysis we investigate how a minimum cycle basis of a graphical structure of the CSP can illustrate complexity. Given a CSP $\mathcal{P}$, a minimum cycle basis MCB of the *incidence graph* of $\mathcal{P}$ is a set of cycles,[1] each represented by a set of variables and constraints. Given a CSP $\mathcal{P}$, a minimum cycle basis MCB_D of the *dual graph* of $\mathcal{P}$ is a set of cycles,[2] each represented by a set of dual variables and dual constraints. Similarly, a minimum cycle basis MCB_{rrD} of the *minimal* dual graph of $\mathcal{P}$ can be defined.

Given an MCB of $\mathcal{P}$, $MCB(x)$ of a variable x is the set of cycles in the MCB in which x appears. Thus,

$$\forall x, \phi \in MCB(x) \Leftrightarrow x \in \phi.$$

---

[1] By extension, we also say the MCB of $\mathcal{P}$.

[2] By extension, we also say the MCB_D of $\mathcal{P}$.

For any given cycle ϕ of an MCB of $\mathcal{P}$, we denote $vertices(\phi)$ the set of variables that appear in ϕ. We denote $vars(MCB(x)) = \cup_{\phi \in MCB(x)} vertices(\phi)$ the set of variables that appear in any cycle of $MCB(x)$.

Given a CSP $\mathcal{P}$, a local consistency property $\mathcal{L}$, and an MCB of $\mathcal{P}$, $\mathcal{P}$ is $\cup_{cycle}\mathcal{L}$ iff every variable X is $\mathcal{L}$ on the subproblem induced by $vars(MCB(x))$. All of these definitions can similarly be defined with MCB_D and MCB_{rrD}.

We consider cactus and block graphs in our theorems.

Definition 15 *A* cactus graph *(sometimes called a cactus tree) is a connected undirected graph in which any two simple cycles have at most one vertex in common. The graph appears as a tree of biconnected components where each component is a simple cycles.*

Definition 16 *A* block graph *(sometimes called a clique tree) is a connected undirected graph in which every biconnected component (block) is a clique. The graph appears as a tree of biconnected components where each component is a simple cycles.*

5.1.2 Binary CSPs

We consider binary CSPs and compute a minimum cycle basis on the constraint graph.[3] We first focus on cactus graphs structures and then block graph structures in the constraint graph.

Cactus-shaped constraint graphs allow $\cup_{cycle}$SAC to find solutions in a backtrack-free manner.

Theorem 2 *If the constraint graph of a binary CSP $\mathcal{P}$ is a cactus graph and $\mathcal{P}$ is $\cup_{cycle}SAC$ consistent, than the domains of $\mathcal{P}$ are minimal.*

---

[3] Indeed, the constraint graph and the incidence graph are the same for binary CSPs.

Sketch of Proof: Each variable-value pair has a partial-solution induced by the biconnected components it participates in. These partial-solutions must be able to be continued to a full solution because the biconnected components only intersect on at most one variable. ☐

We illustrate the above theorem with the following examples:

Example 2 Figure 5.1 shows a CSP with two cycles, intersecting on exactly one variable. This CSP is the 'poster child' for $\cup_{cycle}$SAC-decidable because assigning the intersecting variable creates a tree. All the singleton tests on the articulation node allow us to remove all values that do not participate in any solution. Thus, the problem is $\cup_{cycle}$SAC-decidable.

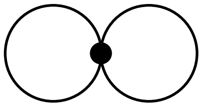


Figure 5.1: A constraint graph with two cyclic biconnected components

Example 3 Figure 5.1 can be extended to any number of cyclic biconnected-components arranged in a tree structure as shown in Figure 5.2. A CSP whose graph has this structure remains $\cup_{cycle}$SAC decidable.

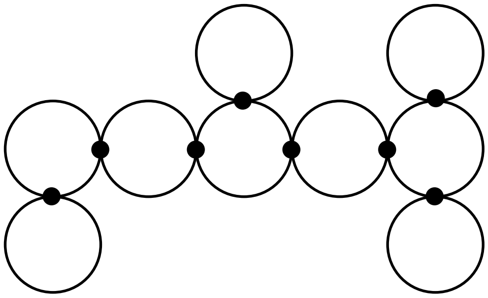


Figure 5.2: A constraint graph that is a tree of cyclic biconnected-components

Example 4 The tree structure of the biconnected components is important. Indeed, examine the constraint graph shown in Figure 5.3 where cycles sharing a single vertex are connected in a cycle. Consider one of the variables at the intersection of two cycles. An MCB of such a variable consists of four cycles: the two small directly adjacent to it, the cycle on the inside (highlighted in gray), and the outer cycle. SAC is not able to break the cycles to determine decidability. Thus, the tree structure of the cycles seems to be an important property.

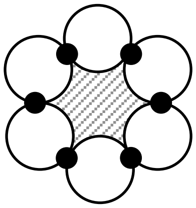
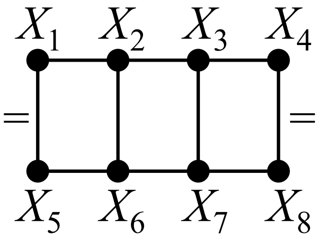


Figure 5.3: A constraint graph made of a cycle of cycles

Example 5 One may wonder whether $\cup_{cycle}$SAC decidable is still guaranteed if we 'changed' a cactus graph by replacing an articulation node between two cycles by a bridge (i.e., an edge whose removal disconnects the graph). Consider the case of a ladder graph. This example is interesting in the sense that the graph considered in every singleton test is a tree. It is thus not unreasonable to wonder whether or not CSP with such a constraint graph can be guaranteed $\cup_{cycle}$SAC decidable. We show that this is not the case with the counterexample shown in Figure 5.4. More generally, the fact that the subgraph induced by a singleton test is a tree does not guarantee $\cup_{cycle}$SAC decidable. The domains of each variable is $\{1, 2, 3, 4\}$. This CSP has no solution but SAC cannot remove any value.

For binary CSPs whose constraint graph is a block graph, we need to increase the level of consistency enforced on the biconnected components.

Theorem 3 *If the constraint graph of a binary CSP $\mathcal{P}$ is a block graph and $\mathcal{P}$ is NIC, then the domains of $\mathcal{P}$ are minimal.*



Scope	Relation
$\langle X_1, X_2 \rangle$	R_1
$\langle X_2, X_3 \rangle$	R_1
$\langle X_3, X_4 \rangle$	R_1
$\langle X_5, X_6 \rangle$	R_2
$\langle X_6, X_7 \rangle$	R_1
$\langle X_7, X_8 \rangle$	R_1
$\langle X_1, X_5 \rangle$	=
$\langle X_2, X_6 \rangle$	R_3
$\langle X_3, X_7 \rangle$	R_3
$\langle X_4, X_8 \rangle$	=

R_1	R_2	R_3
1 3	1 1	1 1
1 4	1 2	1 3
2 1	2 3	2 2
2 2	2 4	2 4
3 1	3 3	3 1
3 2	3 4	3 3
4 3	4 1	4 2
4 4	4 2	4 4

Figure 5.4: A CSP with no solution but SAC removes no values

Sketch of Proof: The neighborhood of every variable is all of the variables in the components (blocks) that it appears in. Because the problem is NIC, there exists a partial solution to every variable-value pair to its neighborhood. By the same argument of Theorem 2 the domains must be minimal. □

However, decidability can be obtained by enforcing NIC on only the articulation points.

Theorem 4 *If the constraint graph of a binary CSP $\mathcal{P}$ is a block graph then enforcing NIC on the variables at the articulation points guarantees decidability.*

Incidentally, note that SAC and POAC are equivalent on cycles. Indeed, consider the network of the binary CSP shown in Figure 5.5. A singleton test on any of

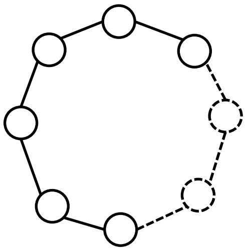


Figure 5.5: The constraint graph of the CSP is a cycle

the variables breaks the cycle into a chain and arc consistency guarantees global consistency [Freuder, 1982].

Proposition 5 *POAC is equivalent to SAC on a cycle.*

5.1.3 Binary and Non-Binary CSPs

In this section, we consider both binary and non-binary CSPs and compute the MCB on the dual graph of the CSP. We state analogous properties for the dual graph and minimal dual graph as was for the constraint graph.

Dual graphs: The consistency properties Singleton Pairwise Consistency (SPWC) and RNIC that operate on the dual graph are analogous to the consistency properties SAC and NIC that operate on the constraint graph.

Theorem 6 *If the dual graph of A CSP $\mathcal{P}$ is a cactus graph and $\mathcal{P}$ is $\cup_{cycle}\mathrm{SPWC}$, then the relations of $\mathcal{P}$ are minimal.*

Theorem 7 *If the dual graph of a CSP $\mathcal{P}$ is a block graph and $\mathcal{P}$ is RNIC, then the relations of $\mathcal{P}$ are minimal.*

Theorem 8 *If the dual graph of a CSP $\mathcal{P}$ is a block graph then enforcing RNIC on the articulation points of the dual graph guarantees decidability.*

The proofs for Theorems 6, 7, and 8 follow from Theorems 2, 3, and 4, respectively.

Minimal dual graphs: In case the original dual graph does is not a cactus or block graph, it may be the case that a *minimal* dual graph has it. (A minimal dual graph is one where redundant edges have been removed.) We denote MCB_{rrD} the set of cycles of a minimal dual graph. To cope with this situation, we need to use $\cup_{cycle_{rrD}}$RNIC, a local consistency property that is strictly stronger than SPWC.

Theorem 9 *If the dual graph of a CSP $\mathcal{P}$ is a cactus graph after removing redundant edges and $\mathcal{P}$ is $\cup_{cycle_{rrD}}$RNIC, then the relations of $\mathcal{P}$ are minimal.*

Theorem 10 *If the dual graph of a CSP $\mathcal{P}$ is a cactus graph after removing redundant edges then enforcing $\cup_{cycle_{rrD}}$*RNIC *on the articulation points of the minimal dual graph guarantees decidability.*

Theorems 9 and 10 follow from Theorems 6 and 8, respectively.

From another perspective, in this situation, $\cup_{cycle_{rrD}}$RNIC is equivalent to *cl*-R($\star$,$|\psi(cl_i)|$)C, where the biconnected components obtained after redundancy removal form the clusters of a tree decomposition [Karakashian *et al.*, 2013].

5.2 Localizing POAC

The algorithm POAC-1, which enforces POAC, runs a singleton test on *each* variable-value pair of the CSP [Balafrej *et al.*, 2014]. In each test, it enforces arc consistency on the *entire* CSP. Whenever the domain of *any* variable is updated, the entire process is repeated (i.e., POAC-1 runs the singleton test on all the variables again). We propose to reduce the cost of POAC-1 in two ways.

1. At a singleton test on a given variable x, we restrict arc consistency to the variables in the cycles in which x appears.

2. Whenever the domain of any variable, x or a variable that appears in a cycle of x, is updated as the result of this test, we repeat the singleton tests on all the variables in the cycles of the affected variable.

Below, we formalize the consistency property NPOAC and $\cup_{cyc}$POAC, that result from our approach, then introduce algorithms NPOACQ and $\cup_{cyc}$POACQ, which implements our idea. Then, we extend, in a trivial manner, our approach to relations. Finally, we discuss the practical improves to the POAC algorithms during search.

5.2.1 NPOAC: Localization to Neighborhoods

We define Neighborhood Partition-One Arc-Consistency (NPOAC) similarly to neighborhood SAC (NSAC) [Wallace, 2015]. Informally, neighborhood POAC localizes the singleton test to the neighborhood of the variable. Given a CSP $\mathcal{P}$ and $\mathcal{V}$ a subset of the variables of $\mathcal{P}$, we denote $\mathcal{P}|_{\mathcal{V}}$ the subproblem induced by $\mathcal{V}$ on $\mathcal{P}$. The constraints included in $\mathcal{P}|_{\mathcal{V}}$ are all those constraints whose scope contains a variable in $\mathcal{V}$.

Definition 17 *A constraint network* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is* Neighborhood Partition-One Arc-Consistent (NPOAC) *iff* $\mathcal{P}$ *is neighborhood SAC (NSAC), and for all* $x_i \in \mathcal{X}$*, for all* $x_j \in neigh(x_i)$*, for all* $v_j \in dom(x_j)$*, there exists* $v_i \in dom(x_i)$ *such that* $v_j \in AC(\mathcal{P}|_{\{x_i\} \cup neigh(x_i)} \cup \{x_i \leftarrow v_i\})$.

Theorem 11 *Neighborhood Inverse Consistency (NIC) is incomparable to Neighborhood POAC (NPOAC).*

Proof: Figure 5.6 shows a CSP that is NPOAC but variable v is not NIC. Figure 5.7 shows a CSP that is NIC but $X_4 \leftarrow 1$ is not NPOAC ($X_4 \leftarrow 1$ is removed in every singleton test for X_1). This example was first proposed to show that POAC is strictly stronger than SAC [Bennaceur and Affane, 2001]. □

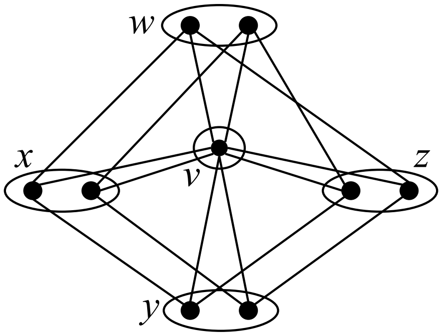


Figure 5.6: NPOAC but not NIC

R_{12}

X_1	X_2
v_1	v_1
v_2	v_2
v_3	v_1
v_3	v_2

R_{13}

X_1	X_3
v_1	v_1
v_1	v_3
v_2	v_2
v_2	v_3
v_3	v_1
v_3	v_2
v_3	v_3

R_{14}

X_1	X_4
v_1	v_1
v_1	v_2
v_2	v_1
v_2	v_2
v_3	v_2

R_{23}

X_2	X_3
v_1	v_2
v_1	v_3
v_2	v_1
v_2	v_3

R_{34}

X_3	X_4
v_1	v_1
v_1	v_2
v_2	v_1
v_2	v_2
v_3	v_2

Figure 5.7: NIC but not NPOAC

5.2.2 $\cup_{cyc}$POAC: Localization to MCBs

For each singleton test for a given variable x_i, we propose to enforce arc consistency on the subproblem induced by the union of the variables of a minimum cycle basis (MCB) of x_i, where a MCB of x_i is computed on the incidence graph of the CSP.

Figure 5.8 shows the incidence graph of a CSP, where the circles denote the variables and the squares the constraints. This graph has three cycles:

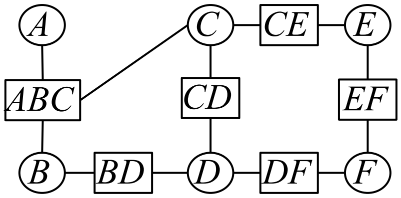


Figure 5.8: A incidence graph

1. (B, ABC, C, CD, D, BD),
2. $(C, CD, D, DF, F, EF, E, CE)$, and
3. $(B, ABC, C, CE, E, EF, F, DF, D, BD)$.

The third cycle can be obtained from the first two by symmetric difference. Thus, the first two cycles constitute a minimal cycle basis for this graph. Incidentally, note that variable A does not appear in any cycle.

We use the same terminology for defining the cycles of a variable as in Section 5.1.1. However, we slightly adjust the definitions of $vars(MCB(x_i))$:

$$vars(MCB(x_i)) = \{x_i\} \cup neigh(x_i) \cup_{\phi \in MCB(x_i)} vertices(\phi).$$

This definition allows us to include variables that do not appear in a cycle (e.g., A does not appear in any cycle in Figure 5.8). This adjustment allows the definition to guarantee arc-consistency.

Given a CSP $\mathcal{P}$ and $\mathcal{V}$ a subset of the variables of $\mathcal{P}$, we denote $\mathcal{P}|_{\mathcal{V}}$ the subproblem induced by $\mathcal{V}$ on $\mathcal{P}$. The constraints included in $\mathcal{P}|_{\mathcal{V}}$ are all those constraints whose scope contains a variable in $\mathcal{V}$.

Now, we formulate the consistency property Union-Cycle Partition-One Arc-Consistency ($\cup_{cyc}$POAC). It is similar to POAC but restricts the propagation of arc consistency during a singleton test for a variable x_i to the subproblem induced on the CSP by the variables in $vars(MCB(x_i))$. Like POAC, the property must hold for all the variables of the CSP.

Definition 18 *Given a minimum-cycle basis MCB of a CSP* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$*, the CSP is* Union-Cycle Partition-One Arc-Consistent ($\cup_{cyc}$POAC) *iff* $\forall x_i \in \mathcal{X}$*, the CSP* $\mathcal{P}$ *is AC for all* $v_i \in dom(x_i)$ *on* $\mathcal{P}|_{vars(MCB(x_i))} \cup \{x_i \leftarrow v_i\}$*, and* $\forall x_{j \neq i} \in \mathcal{X}, v_j \in dom(x_j), \exists v_i \in dom(x_i)$ *such that* $v_j \in AC(\mathcal{P}|_{vars(MCB(x_i))} \cup \{x_i \leftarrow v_i\})$.

It is easy to see that $\cup_{cyc}$POAC is strictly stronger than GAC, not comparable with SAC, and strictly weaker than POAC.

5.2.3 NPOACQ: A Variable-Based Algorithm

POAC-1, the original algorithm for POAC, uses a list of all the CSP variables, ordered by some heuristic such as decreasing values of dom/wdeg [Balafrej *et al.*, 2014]. After processing once every variable in the list, it repeats the process again whenever any domain is updated. Importantly, POAC-1 does not reconsider any variable for singleton testing before all the variables of the CSP have been processed. The size of the list does not change. To implement a similar behavior, our algorithm NPOACQ (Algorithm 5) uses three queues: Q stores the variables to be processed by singleton testing, Q_{seen} stores the variables that have been processed during the current iteration, and $Q_{toRevisit}$ stores the variables affected by change during the current iteration. Only when all the variables in Q have been processed (Q is empty), the variables in $Q_{toRevisit}$ are moved to Q to be processed.

Q is handled as a priority list using the same heuristic as POAC-1 (Line 4 of Algorithm 5). The popped variable is stored in Q_{seen} (Line 5) so that no variable is re-processed for singleton testing before Q is empty. `var`NPOACQ (Algorithm 7) is then called (Line 6 of Algorithm 5) to execute singleton tests for the popped variable. In Lines 9 and 19, `var`NPOACQ calls ReQueue (Algorithm 6) on all the variables in the neighborhood of any variable whose domain was updated. ReQueue adds those variables to $Q_{toRevisit}$ in case they were already singleton tested during the current iteration (Line 1), otherwise it adds them to Q (Line 2). When Q is empty, the variables in $Q_{toRevisit}$ are moved to Q, and Q_{seen} is cleared (Lines 7 and 8 of Algorithm 5).

`var`NPOACQ (Algorithm 7) runs singleton tests on a given CSP variable by calling TestAC (Algorithm 8) which enforces arc consistency on the subproblem induced on the CSP by the variables in $neigh(x_i)$ (Line 4). As in POAC-1, whenever a value

```
Algorithm 5: NPOACQ($\mathcal{P}$)
   Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A CSP instance
   Output: $true$ when $\mathcal{P}$ is $\cup_{cyc}$POAC, otherwise $false$
1  $Q \leftarrow \mathcal{V}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
2  $consistent \leftarrow$ EnforceAC($\mathcal{P}, \emptyset$)
3  while $consistent$ and $Q \neq \emptyset$ do
4      $x_i \leftarrow$ Pop($Q$)
5      $Q_{seen} \leftarrow Q_{seen} \cup \{x_i\}$
6      if not varNPOACQ($x_i, \mathcal{P}$) then return $false$
7      if $Q = \emptyset$ and $Q_{toRevisit} \neq \emptyset$ then
8          $Q \leftarrow Q_{toRevisit}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
9  return $true$
```

```
Algorithm 6: ReQueue($x_i$)
   Input: $x_i$: a variable to requeue
   Output: Adds $x_i$ to either $Q$ or $Q_{toRevisit}$
1  if $x_i \in Q_{seen}$ then $Q_{toRevisit} \leftarrow Q_{toRevisit} \cup \{x_i\}$
2  else $Q \leftarrow Q \cup \{x_i\}$
```

is removed from a variable's domain, `var`NPOACQ enforces AC on the CSP (Lines 6 and 20).

Like POAC-1, we use the data structure $counter(\cdot,\cdot)$. $counter(x_j, v_j)$ records how many times value v_j of variable x_j was pruned during the singleton tests for another variable x_i. If, after running all the singleton tests for x_i, $counter(x_j, v_j) = |dom(x_i)|$, then we know that (x_j, v_j) is necessarily inconsistent and can be safely removed. TestAC (Algorithm 8) implements the singleton test for $x_i \leftarrow v_i$ and updates $counter(\cdot,\cdot)$. EnforceAC($\mathcal{P}, L$) allows running *any* arc consistency algorithm. It stores in L the list of variable-value pairs that were removed as a result of enforcing AC. TestAC (Algorithm 8) updates the counters only when the problem is arc consistent (Line 6).

Algorithm 7: `var`NPOACQ($x_i, \mathcal{P}$)

```
Input: $x_i$: Variable to instantiate; $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A CSP instance; $Q$: The propagation queue
Output: $true$ if consistent, else $false$
$\forall x_j \in \mathcal{V}, v_j \in dom(x_j), counter(x_j, v_j) \leftarrow 0$
$size \leftarrow |dom(x_i)|$
foreach $v_i \in dom(x_i)$ do
    if not TestAC($\{x_i\} \cup neigh(x_i), \mathcal{D}, cons(x_i) \cup \{x_i \leftarrow v_i\}, counter(\cdot,\cdot)$) then
        $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
        if not EnforceAC($\mathcal{P}, L$) then return $false$
if $dom(x_i) = \emptyset$ then return $false$
if $|dom(x_i)| \neq size$ then
    foreach $x_k \in neigh(x_i)$ do ReQueue($x_k$)
$change \leftarrow false$
foreach $x_j \in neigh(x_i)$ do
    $size \leftarrow |dom(x_j)|$
    foreach $v_j \in dom(x_j)$ do
        if $counter(x_j, v_j) = |dom(x_i)|$ then
            $dom(x_j) \leftarrow dom(x_j) \setminus \{v_j\}$, $change \leftarrow true$
        $counter(x_j, v_j) \leftarrow 0$
    if $dom(x_j) = \emptyset$ then return $false$
    if $|dom(x_j)| \neq size$ then
        foreach $x_k \in neigh(x_j) \setminus \{x_i\}$ do ReQueue($x_k$)
if change and not EnforceAC($\mathcal{P}, \emptyset$) then return $false$
return $true$
```

Algorithm 8: TestAC($\mathcal{P}, counter(\cdot,\cdot)$)

```
Input: $\mathcal{P}$: A CSP instance; $counter(\cdot,\cdot)$: the counter data structure
Output: $true$ if consistent, else $false$
$L \leftarrow \emptyset$
$consistent \leftarrow$ EnforceAC($\mathcal{P}$, $L$)
foreach $(x_j, v_j) \in L$ do
    $dom(x_j) \leftarrow dom(x_j) \cup \{v_j\}$
    if $consistent$ then
        $counter(x_j, v_j) \leftarrow counter(x_j, v_j) + 1$
return $consistent$
```

5.2.4 $\cup_{cyc}$POACQ: A Variable-Based Algorithm

$\cup_{cyc}$POACQ (Algorithm 9) is similar to NPOACQ (Algorithm 5). The major difference is in Line 6, where `var`$\cup_{cyc}$POACQ (Algorithm 10) is called to execute singleton tests for the popped variable.

`var`$\cup_{cyc}$POACQ (Algorithm 10) is similar to `var`NPOACQ (Algorithm 7), which runs singleton tests on a given CSP. The major difference is in Line 4, where TESTAC is induced on the MCB of a variable, rather than its neighborhood. `var`$\cup_{cyc}$POACQ does not restrict how MCBs are generated (i.e., using exact or approximate algorithms) or the graphs (i.e., incidence or dual) on which they are computed.

Algorithm 9: $\cup_{cyc}$POACQ($\mathcal{P}, MCB$)

```
   Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A CSP instance; $MCB$: a minimum cycle basis of $\mathcal{P}$
   Output: $true$ when $\mathcal{P}$ is $\cup_{cyc}$POAC, otherwise $false$
1  $Q \leftarrow \mathcal{V}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
2  $consistent \leftarrow$ ENFORCEAC($\mathcal{P}, \emptyset$)
3  while $consistent$ and $Q \neq \emptyset$ do
4      $x_i \leftarrow$ POP($Q$)
5      $Q_{seen} \leftarrow Q_{seen} \cup \{x_i\}$
6      if not var$\cup_{cyc}$POACQ($x_i, \mathcal{P}, MCB$) then
7          return $false$
8      if $Q = \emptyset$ and $Q_{toRevisit} \neq \emptyset$ then
9          $Q \leftarrow Q_{toRevisit}$, $Q_{toRevisit} \leftarrow \emptyset$, $Q_{seen} \leftarrow \emptyset$
10 return $true$
```

5.2.5 Extension to Relations

We extend the definition of POAC to relations.

Definition 19 *A CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is* Relational Partition-One Arc-Consistent (rPOAC) *iff the CSP is singleton PWC, and for all $c_i \in \mathcal{C}$, for all $\tau_i \in R_i$, for all $c_j \in \mathcal{C}$, there exists $\tau_j \in R_j$ such that $(c_i, \tau_i) \in PWC(\mathcal{P} \cup \{R_i \leftarrow \tau_i\})$.*

Algorithm 10: var$\cup_{cyc}$POACQ($x_i, \mathcal{P}, MCB$)

```
Input: $x_i$: Variable to instantiate; $\mathcal{P}$: A CSP instance; $MCB$: a minimum cycle
       basis of $\mathcal{P}$
Output: $true$ if consistent, else $false$
$\forall x_j \in \mathcal{V}, v_j \in dom(x_j), counter(x_j, v_j) \leftarrow 0$
$size \leftarrow |dom(x_i)|$
foreach $v_i \in dom(x_i)$ do
    if not TestAC($\mathcal{P}|_{vars(MCB(x_i))} \cup \{x_i \leftarrow v_i\}, counter(\cdot,\cdot)$) then
        $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
        if not EnforceAC($\mathcal{P}, \emptyset$) then return $false$
if $dom(x_i) = \emptyset$ then return $false$
if $|dom(x_i)| \neq size$ then
    foreach $x_k \in vars(MCB(x_i)) \setminus \{x_i\}$ do ReQueue($x_k$)
$change \leftarrow false$
foreach $x_j \in vars(MCB(x_i)) \setminus \{x_i\}$ do
    $size \leftarrow |dom(x_j)|$
    foreach $v_j \in dom(x_j)$ do
        if $counter(x_j, v_j) = |dom(x_i)|$ then
            $dom(x_j) \leftarrow dom(x_j) \setminus \{v_j\}$, $change \leftarrow true$
        $counter(x_j, v_j) \leftarrow 0$
    if $dom(x_j) = \emptyset$ then return $false$
    if $|dom(x_j)| \neq size$ then
        foreach $x_k \in vars(MCB(x_j)) \setminus \{x_j\}$ do ReQueue($x_k$)
if $change$ and not EnforceAC($\mathcal{P}, \emptyset$) then return $false$
return $true$
```

The property rPOAC can be extended to Relational Neighborhood-POAC (rNPOAC), and Relational Union-Cycle POAC ($\cup_{cyc}$POAC).

Theorem 12 *Relational Neighborhood Inverse Consistency (RNIC) is incomparable to Relational Neighborhood POAC (rNPOAC).*

Proof: Follows from Theorem 11. □

The algorithm to enforce rPOAC is a trivial adaptation of the variable-based POAC algorithm: it operates on relations' tuples instead of variables' values. Further, instead of AC, we enforce pair-wise consistency (PWC). We denote rNPOACQ and $\cup_{cyc}$rPOACQ the adaptation of NPOACQ and $\cup_{cyc}$POACQ, respectively, to relations.

Preliminary studies enforcing rPOAC showed that enforcing PW-AC2 is, in general, faster and can solve the most benchmarks than the rPOAC variants [Woodward *et al.*, 2016a]. Combining the adaptive mechanism of APOAC [Balafrej *et al.*, 2014] with rPOAC proved to be beneficial, but still did not outperform PW-AC2. We strongly believe that the adaptive mechanism could be further improved with better tuning of the parameters, which is beyond the topic of this dissertation. Further, improving the PWC algorithm will boost the performance of rPOAC algorithms. It is too early to rule out the usefulness of the relational versions of POAC, the effectiveness of propagation over cycles is noteworthy even in this context.

5.2.6 Practical Improvement of Algorithms

Below, we make useful observations for improving the performance of the POAC algorithm in practice.

Singleton domains. Because the algorithms for enforcing POAC-like properties (e.g., POAC-1 and $\cup_{cyc}$POACQ) enforce GAC whenever singleton testing a variable yields a domain update, variables with a singleton domain never need to be singleton tested. We do not include this test in our pseudocode to avoid reducing readability.

Proposition 13 *On a CSP that is GAC, singleton testing a variable* x *with* $|dom(x)| = 1$ *yields no filtering.*

Proof: After assigning x to the unique value in its domain, the CSP remains GAC. □

Domino effect. This observation allows us, during backtrack search using a POAC-like algorithm for real-full lookahead, to instantiate all variables with singleton domains (i.e., domino effect) without re-enforcing consistency because no further filtering can be obtained, thus saving on effort. Note that the same behavior is implicitly guaranteed for consistency algorithms using supports.

Q initialization. After an assignment $x \leftarrow v$, Q is initialized to $neigh(x)$ for NPOAC and $vars(MCB(x)) \setminus \{x\}$ for $\cup_{cyc}$POAC.

Large variables' domains. On small variables' domains singleton testing is *quicker* and empirically yields *more filtering* than on larger variables' domains. The observation explains why using dom/wdeg to order to the variables for singleton testing yields good performance in practice. It also explains the good performance of the adaptive algorithm APOAC, which avoids singleton testing variables with large domains, a costly process that rarely yields any filtering.

5.3 Approximating a Minimum Cycle Basis

The time complexity of the exact algorithm for computing a minimum cycle basis (MCB) is $\mathcal{O}(e^2 n/\log(n))$ where n is the number of vertices and e the number of edges in the graph [Amaldi *et al.*, 2010]. The approximate algorithm for computing an MCB is $\mathcal{O}(e^\omega \sqrt{n \log(n)})$ where ω is the best exponent of matrix multiplication ($\omega < 2.376$) [Kavitha *et al.*, 2007].

We first give an evaluation of a minimum cycle basis, showing that it takes too much time in practice. Then we give our approximation, followed by an empirical comparison to computing an MCB.

5.3.1 Minimum Cycle Basis Evaluation

On some problems computing a minimum cycle basis using either the exact algorithm [Mehlhorn and Michail, 2009; Amaldi *et al.*, 2010] or the approximate algorithm [Kavitha *et al.*, 2007] takes more time and memory than we are willing to give it (i.e., greater than a minute and 8GB). We compute a minimum cycle basis using the algorithm of Amaldi *et al.* [2010], and Table 5.1 shows:

- The total number of instances (#Instances).
- The number of instances that could compute a MCB (#Completed).
- The number of instances that timed out while computing a MCB (#Time Out).
- The number of instances that reached the memory limit while computing a MCB (#Mem Out).
- The average memory consumption to load the CSP, initialize POAC, and compute the cycles (Avg. Memory [MB]).
- The average time to compute the minimum cycle basis (Avg. Time [sec]), without solving the problem. For a comparison, the average time to find the first solution using GAC is given in parenthesis.
- The average size differences of the resulting *neighborhoods* for each variable in $\cup_{cyc}$POAC (Vertices beyond neighborhood).

On all the benchmarks the average memory usage was 709.3 and the average time to compute the cycles was 163.4 seconds. We report a selection of benchmarks where the performance of computing an MCB differs greatly. For the jobShop-ewddr2 and myciel benchmarks computing a MCB has minimal overhead. These benchmarks showcase an ideal situation of computing the MCB.

Table 5.1: Time and memory to compute a minimum cycle basis

Benchmark	**#Instances**	**#Completed**	**#Time Out**	**#Mem Out**	**Avg. Memory [MB]**	**Avg. Time [sec]**	**Vertices beyond neighborhood**
All Benchmarks	3,525	2,851	19	655	709.3	163.4 (924.0)	7.9
jobShop-ewddr2	10	10	0	0	767.0	0.1 (13.0)	3.5
myciel	16	16	0	0	580.7	54.9 (872.6)	7.6
QCP-20	15	15	0	0	7,667.3	2,905.5 (2,755.2)	3.6
ehi-90	100	100	0	0	4,109.5	1,549.0 (3.7)	0.0
domino	16	12	1	3	341.9	5.5 (59.7)	597.0
full-insertion	41	22	0	19	1,209.3	153.1 (80.9)	4.1

However, not all benchmarks elicit good performance computing an compute the MCB. On the QCP-20 and ehi-90 benchmarks, computing a MCB requires a large amount of memory and CPU time. These benchmarks could be solved using GAC faster than the time it took to compute a MCB, which does not start the solving process. Further, for ehi-90, the extra computation resulted in no gain of the neighborhood size. The domino and full-insertion benchmarks hit the memory limit while computing a MCB.

5.3.2 Approximation Cycles Using a Breath-First Search

Because of the poor time and memory consumption of computing a minimum cycle basis, we introduce a new approximation. Our approximation does not guarantee the minimum cycle basis property nor does it compute a basis. Instead, the approximation heuristically finds local cycles for every variable in the problem, which is the central goal for using the cycles in $\cup_{cyc}$POAC. Algorithm 11 presents our algorithm for finding the cycles of each variable. We call this algorithm BFSC as we are conducting a breath-first search (BFS) to find the cycles of the graph. RootedBFSC (Algorithm 12) is called on every variable in the problem to find cycles involving that variable (i.e., node in the graph).

Algorithm 11: BFSC($\mathcal{P}$)

```
Input: $\mathcal{X}$: The set of variables in the CSP
Output: $allCycles$: A set of detected cycles in the problem
$allCycles \leftarrow \emptyset$
foreach $x \in \mathcal{X}$ do
    $allCycles \leftarrow allCycles \cup$ RootedBFSC($x$)
return $allCycles$
```

Algorithm 12 (RootedBFSC) conducts the breath first search starting from a given node in the graph. The breath first search attempts to find the shallowest cycle that involves every node in the neighborhood of the root node. We use two maps, *seenNodesFrom* and *seenNodeParent*, to record what neighbor a node was first visited from and their parents in the breath first search, respectively. Note that *seenNodesFrom* can be obtained by traversing *seenNodeParent* until a neighbor of *root* is reached, but we choose to record it in its own data-structure to save on this operation. Initially the only node we have seen is the *root* node who has no parent (Line 3).

Our heuristic attempts to find one cycle for every neighbor of *root*. We store in *neighToMatch* a set of neighbors that we still need to find a cycle for, initially all neighbors of root (Line 7). We record that we have seen all of the neighbors, and that their parents are root (Line 8). We start the breath first search from the neighbors by inserting them into the list of nodes to visit *toVisit* (Line 9).

We pop from the front of the *toVisit* (Line 11), and if this *node* is rooted from a neighbor that we need to match, we attempt to see if we can form a cycle to any of the

Algorithm 12: RootedBFSC(*root*)

```
Input: $root$: A root node to run BFS on
Output: $cycles$: A set of cycles
$cycles \leftarrow \emptyset$
$seenNodesFrom \leftarrow \emptyset$; $seenNodesParent \leftarrow \emptyset$
$seenNodesFrom[root] \leftarrow \perp$; $seenNodesParent[root] \leftarrow \perp$
$toVisit \leftarrow []$
$neighToMatch \leftarrow \emptyset$
foreach $neigh \in neigh(root)$ do
    $neighToMatch \leftarrow neighToMatch \cup \{neigh\}$
    $seenNodesFrom[neigh] \leftarrow neigh$; $seenNodesParent[neigh] \leftarrow root$
    $toVisit \leftarrow$ PushBack($neigh, toVisit$)
while $toVisit \neq \emptyset$ do
    $node \leftarrow$ PopFront($toVisit$)
    if $seenNodesFrom[node] \in neighToMatch$ then
        foreach $neigh \in neigh(node)$ do
            if $neigh \notin seenNodesFrom$ then
                $seenNodesFrom[neigh] \leftarrow node$
                $seenNodesParent[neigh] \leftarrow seenNodesParent[node]$
                PushBack($neigh, toVisit$)
            else
                if $seenNodesFrom[node] \neq seenNodesFrom[neigh]$ then
                    $neighToMatch \leftarrow neighToMatch \setminus$
                      $\{seenNodesFrom[node], seenNodesFrom[neigh]\}$
                    $cycle \leftarrow [node, neigh]$
                    $visitedNode \leftarrow node$
                    while $visitedNode \neq root$ do
                        PushFront($visitedNode, cycle$)
                        $visitedNode \leftarrow seenNodesParent[visitedNode]$
                    $visitedNode \leftarrow neigh$
                    while $visitedNode \neq root$ do
                        PushBack($visitedNode, cycle$)
                        $visitedNode \leftarrow seenNodesParent[visitedNode]$
                    PushBack($root, cycle$)
                    $cycles \leftarrow cycles \cup \{cycle\}$
return $cycles$
```

neighbors of *root*. To find a cycle, we check each neighbor of *node* (Line 13). If the neighbor has not been visited, we populate its *seenNodesFrom* and *seenNodesParent* and add it to the list of nodes to visit (Lines 15–17). Otherwise, the neighbor has been visited before, and if it was discovered from different neighbors of *root*, we can form a cycle (Line 19).

If we formed a cycle, we stop processing nodes from this cycle (Line 20). We form the cycle between *node* and *neigh* (Line 21) by traversing the *seenNodesParent* until we reach *root* (Lines 22–29). We then add the *root* to the cycle (Line 30) and add the cycle to the set of all cycles (Line 31).

The time complexity of the BFSC algorithm n calls to RootedBFSC, which is $\mathcal{O}(n \cdot e)$, where n is the number of variables and e is the number of edges. This complexity is smaller than the time complexity of the exact algorithm for computing an MCB, $\mathcal{O}(e^2 n/\log(n))$ [Amaldi *et al.*, 2010].

5.3.3 Comparing Cycles Found by BFSC and MCB

Table 5.2 shows the result of computing the cycles using the approximation BFSC and the MCB algorithm of Amaldi *et al.* [2010] (MCB), showing the same information as Table 5.1. Overall, we can compute cycles on more instances using the BFSC algorithms than MCB. We can compute on more instances because we reduce the number of instances that memout, and can its computation on each instances is much quicker. On the QCP-20 and ehi-90 benchmarks BFSC uses an order of magnitude less memory. However, our approximation does not find any cycles. Indeed, with BFSC there is no increase in the neighborhood sizes, while the MCB had 3.6 variables beyond the neighborhood.

Table 5.2: Comparing computing cycles using MCB and BFSC

Benchmark	#Instances	#Completed			#TimeOut		#MemOut		Avg. Mem [MB]		Avg. Time [sec]		Vertices beyond neighborhood	
		By One	MCB	BFSC	MCB	BFSC	MCB	BFSC	MCB	BFSC	MCB	BFSC	MCB	BFSC
Summary	3,525	2,851	2,851	3,082	19	6	655	437	709.3	184.6	163.4	2.3	7.9	8.8
jobShop-ewddr2	10	10	10	10	0	0	0	0	767.0	767.1	0.1	0.0	3.5	4.3
myciel	16	16	16	16	0	0	0	0	580.7	37.4	54.9	0.5	7.6	9.0
QCP-20	15	15	15	15	0	0	0	0	7,667.3	505.7	2,905.5	26.6	3.6	0.0
ehi-90	100	100	100	100	0	0	0	0	4,109.5	206.7	1,549.0	2.9	0.0	0.0
domino	16	12	12	12	1	1	3	3	341.9	226.5	5.5	1.6	597.0	597.0
full-insertion	41	22	22	37	0	0	19	4	1,209.3	66.6	153.1	0.4	4.1	12.8

5.4 Empirical Evaluation

The goal of the section is to assess the effectiveness of localizing POAC to neighborhoods and cycles when used for real-full lookahead during search. To that end, we evaluate finding a single solution to a CSP using backtrack search, real-full lookahead, and the dom/wdeg variable ordering heuristic [Boussemart *et al.*, 2004].

We first discuss our experimental setup. The adaptive POAC of Balafrej *et al.* [2014] is a *how much* strategy, terminating the POAC-1 algorithm early. We evaluate combining this strategy with our *where* strategies of localizing POAC to neighborhoods and cycles. We then evaluate the PrePeak$^+$ strategy of Chapter 4, which is a when and how much strategy, combined with localizing POAC.

5.4.1 Experimental Setup

We set up our experiments as follows. We use STR2+ [Lecoutre, 2011] as the GAC algorithm for lookahead. We use the POAC-1 algorithm [Balafrej *et al.*, 2014] for enforcing POAC. We also evaluate using NPOACQ (Section 5.2.3, Algorithm 5) and $\cup_{cyc}$POACQ (Section 5.3, Algorithm 9). We compute the cycles from a minimum cycle basis using the algorithm of Amaldi *et al.* [2010] ($\cup_{cyc}^{mcb}$POACQ) or approximated by BFSC of Section 5.3 ($\cup_{cyc}^{bfsc}$POACQ). For all the POAC algorithms we use dom/wdeg to select the variable for singleton testing. In particular, this orders the list of variables to process in POAC-1 and the propagation queue for the POACQ-based algorithms.

In general, POAC is too expensive to enforce until quiescence. Balafrej *et al.* [2014] advocated for an adaptive version of POAC (APOAC), which is a 'how much' strategy that interrupts the singleton testing after a given number of variables has been processed. This cutoff values is learned during search. We use the best adaptive version reported by Balafrej *et al.* [2014], where the maximum number of singleton calls, $maxK$, is initialized to the number of variables in the problem. The algorithm spends 1/10 of its time learning[4] a $maxK$ threshold and 9/10 of its time exploiting the learned $maxK$. In our experiments, we evaluate using the adaptive versions, which is denoted by adding an 'A' before all the algorithm names.

We conducted the experiments on the following benchmark problems from Lecoutre's webpage:[5] including all benchmarks with at least one instance with a primal graph of density less than 50%.[6] We set a time limit of 60 minutes per instance with 8GB of memory.

---

[4] Using the terminology of Balafrej *et al.* [2014], $maxK = n$, last drop with $\beta = 0.05$, and 70%-PER.

[5] `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

[6] Table E.1 in Appendix E list the selected benchmarks.

5.4.2 Localizing Adaptive POAC

Table 5.3 compares the performance of the method for finding the cycles: a minimum cycle basis (MCB) or the approximation using a BFS (BFSC, Section 5.3). A$\cup_{cyc}^{bfsc}$POAC solves more instances than A$\cup_{cyc}^{mcb}$POAC, showing that our approximation technique is useful in this context. Notice that the average node visits of A$\cup_{cyc}^{bfsc}$POAC is smaller than A$\cup_{cyc}^{mcb}$POAC, which is expected as because BFSC computed larger neighborhoods for each variable than MCB (i.e., a larger vertices beyond neighborhood in Table 5.2). Thus, in the remaining experiments, we will only report the results of the cycles computed by BFSC.

Table 5.3: Comparing lookahead using A$\cup_{cyc}^{bfsc}$POAC and A$\cup_{cyc}^{mcb}$POAC

Algorithm	**A$\cup_{cyc}^{bfsc}$POAC**	**A$\cup_{cyc}^{mcb}$POAC**
#solved	**2,217**	2,157
∑CPU [sec]	>**500,767.3**	>853,730.6
avg. NV	163,525.9	177,010.3
#Instances 3,525 total, 2,150 by all, 2,224 by at least one		

Table 5.4 compares the performance of APOAC, ANPOAC, and A$\cup_{cyc}^{bfsc}$POAC. APOAC performs the worst by solving the fewest number of instances in the largest CPU time. However, it was also the strongest consistency enforced and resulted in the small number of node visits. This result is not surprising as we saw when evaluating our triggering strategy in Section 4.4. Localizing POAC to the neighborhoods and

Table 5.4: Lookahead with adaptive POAC techniques

Algorithm	**GAC**	**APOAC**	**ANPOAC**	**A$\cup_{cyc}^{bfsc}$POAC**
#solved	**2,277**	2,132	2,224	2,215
∑CPU [sec]	>**368,158.3**	>1,102,942.7	>659,937.3	>763,566.4
avg. NV	460,387.0	**15,348.0**	349,452.9	66,373.8
#Instances 3,525 total, 2,111 by all, 2,295 by at least one				

cycles (i.e., ANPOAC and A$\cup_{cyc}$POAC) solve more instances than APOAC, but not as many as GAC. Thus, localizing POAC is not enough to overcome GAC by itself.

Table 5.5 investigates a selection of benchmarks where APOAC performed well, to help identify what is happening with the localizations. For the dubois, pseudo-ii, mug, and k-insertion benchmarks, APOAC performs the best on all measures, while ANPOAC is the worst adaptive-POAC technique in terms of CPU time and

Table 5.5: APOAC techniques on select benchmarks where APOAC beats GAC

Algorithm	**GAC**	**APOAC**	**ANPOAC**	**A$\cup_{cyc}$POAC**
dubois #Instances 13 total, 6 by all, 11 by at least one				
#solved	6	**11**	6	8
$\sum$CPU [sec]	>22,284.0	**2,233.3**	>23,162.1	>14,456.3
avg. NV	123,405,942.7	528,813.3	85,514,422.2	7,560,067.3
pseudo-ii #Instances 41 total, 9 by all, 14 by at least one				
#solved	9	**14**	9	9
$\sum$CPU [sec]	>18,619.8	**2,481.9**	>18,705.1	>18,592.0
avg. NV	282,191.0	**72,442.0**	282,191.0	139,305.0
mug #Instances 8 total, 4 by all, 6 by at least one				
#solved	4	**6**	5	**6**
$\sum$CPU [sec]	>7,200.1	**2,974.8**	>4,005.4	3,168.0
avg. NV	94.0	94.0	94.0	94.0
k-insertion #Instances 32 total, 16 by all, 18 by at least one				
#solved	16	**18**	16	17
$\sum$CPU [sec]	>7,554.1	**4,177.6**	>7,632.3	>5,360.2
avg. NV	367,761.6	**14,763.0**	367,761.6	92,015.3
cril #Instances 8 total, 3 by all, 7 by at least one				
#solved	4	6	**7**	**7**
$\sum$CPU [sec]	>12,655.6	>4,199.3	2,465.1	**2,419.3**
avg. NV	367,754.3	**107,020.3**	197,697.3	135,826.3
QWH-20 #Instances 10 total, 9 by all, 9 by at least one				
#solved	**9**	**9**	**9**	**9**
$\sum$CPU [sec]	2,581.3	1,870.9	**1,290.4**	1,522.1
avg. NV	532,566.1	**33,813.4**	137,518.7	137,518.7
QCP-20 #Instances 15 total, 4 by all, 5 by at least one				
#solved	4	4	**5**	**5**
$\sum$CPU [sec]	>5,328.7	>4,861.0	**4,557.6**	4,696.9
avg. NV	770,529.5	**44,462.8**	568,227.8	568,227.8

A$\cup_{cyc}$POAC's CPU time is between the APOAC and ANPOAC. On the pseudo-ii benchmark, ANPOAC has the exact same search tree as GAC, which shows that the neighborhood is too localized to offer filtering. For the cril benchmark, A$\cup_{cyc}$POAC has the smallest CPU time, thus finding cycles can perform best. On the QWH-20 and QCP-20 benchmarks, ANPOAC has the smallest CPU time. The BFSC method of finding cycles did not identify any cycles that 'grew' a neighborhood, thus A$\cup_{cyc}$POAC is equivalent to ANPOAC.

Table 5.6 investigates a selection of benchmarks where APOAC performed poorly, to help identify what is happening with the localizations. For these benchmarks, GAC performs the best over all adaptive POAC techniques. ANPOAC is second in terms of CPU time, while A$\cup_{cyc}$POAC and APOAC are third and fourth, respectively. On these benchmarks even enforcing a localized version of POAC is detrimental. This explains why looking at all benchmarks (Table 5.4) ANPOAC has a smaller CPU

Table 5.6: APOAC techniques on select benchmarks where GAC beats APOAC

Algorithm	**GAC**	**APOAC**	**ANPOAC**	**A$\cup_{cyc}$POAC**
geom #Instances 100 total, 98 by all, 100 by at least one				
#solved	**100**	98	**100**	**100**
$\sum$CPU [sec]	**7,254.3**	>28,365.6	14,427.7	14,433.1
avg. NV	20,915.3	3,373.0	5,213.8	5,217.5
tightness0.35 #Instances 100 total, 100 by all, 100 by at least one				
#solved	**100**	**100**	**100**	**100**
$\sum$CPU [sec]	**7,547.1**	32,155.0	11,120.8	11,487.5
avg. NV	80,978.6	**9,901.0**	58,410.6	57,073.3
super-os-taillard-4 #Instances 30 total, 22 by all, 28 by at least one				
#solved	**28**	22	**28**	**28**
$\sum$CPU [sec]	**7,647.2**	>33,042.7	17,170.5	17,185.3
avg. NV	8,416.7	**17.0**	114.4	114.4
wordsVg #Instances 65 total, 58 by all, 65 by at least one				
#solved	**65**	58	**65**	63
$\sum$CPU [sec]	**8,402.3**	>37,232.1	8,782.8	>27,807.3
avg. NV	**17,133.3**	1,015.9	17,141.7	9,620.0

time than A$\cup_{cyc}$POAC but larger than GAC. The majority of the benchmarks are those where GAC performs best. Thus, the weakest POAC technique will have the second smallest CPU time.

5.4.3 Combining PrePeak$^+$ and Localized POAC

From the evaluations localizing adaptive-POAC (Section 5.4.2), we saw that $\cup_{cyc}$POAC provides a compromise between POAC and NPOAC, but the technique is detrimental because of instances where enforcing POAC, and its localized versions, are not beneficial. The goal of PrePeak$^+$ (Chapter 4) is to determine the usefulness of a higher-level consistency and enforce it accordingly. In this section we use the localized versions of POAC, NPOAC and $\cup_{cyc}$POAC, with that of the triggering scheme PrePeak$^+$.

Table 5.7 compares the performance of PrePeak$^+$ with triggering POAC techniques. Combining PrePeak$^+$ with NPOAC and $\cup_{cyc}$POAC is worse than GAC in terms of completions and CPU time. Both NPOAC and $\cup_{cyc}$POAC have similar average node visits to that of GAC. Given the relatively few calls to POAC for NPOAC and $\cup_{cyc}$POAC (20.8 and 31.8) compared with POAC (256.5), it makes sense there is not a large reduction of node visits. Indeed, because NPOAC and $\cup_{cyc}$POAC

Table 5.7: PrePeak$^+$ with POAC techniques

		PrePeak$^+$		
Algorithm	**GAC**	**POAC**	**NPOAC**	**$\cup_{cyc}$POAC**
#solved	2,277	**2,284**	2,276	2,273
$\sum$CPU [sec]	>350,158.3	>**323,045.3**	>358,297.3	>359,930.5
avg. NV	511,621.9	**229,898.3**	534,553.4	506,888.8
#CallsPOAC	-	256.5	20.8	31.8
#Instances 3,525 total, 2,268 by all, 2,290 by at least one				

are weaker consistencies, PrePeak$^+$ learns that they are not effective (i.e., causing domain wipeouts as frequently) and does not trigger them very often.

We now focus on individual benchmarks to help show our reasoning as to why PrePeak$^+$ with POAC performs better than PrePeak$^+$ with NPOAC and $\cup_{cyc}$POAC. Table 5.8 shows a selection of benchmarks where PrePeak$^+$ with POAC performs the best. For dubois and k-insertion benchmarks, the number of calls to POAC is large for PrePeak$^+$ with POAC. However, PrePeak$^+$ with NPOAC and $\cup_{cyc}$POAC

Table 5.8: Benchmarks where PrePeak$^+$ with POAC performs well

		PrePeak$^+$		
Algorithm	**GAC**	**POAC**	**NPOAC**	$\cup_{cyc}$**POAC**
dubois #Instances 13 total, 6 by all, 11 by at least one				
#solved	6	**11**	6	6
$\sum$CPU [sec]	>22,284.0	**4,157.8**	>24,447.9	>24,120.7
avg. NV	123,405,942.7	**24,440,578.0**	132,236,419.2	125,963,931.7
#CallsPOAC	-	54,073.8	17.0	22.3
k-insertion #Instances 32 total, 16 by all, 17 by at least one				
#solved	16	**17**	16	16
$\sum$CPU [sec]	>3,954.1	**2,839.7**	>3,982.4	>4,039.4
avg. NV	367,761.6	93,532.0	367,761.6	422,957.0
#CallsPOAC	-	1,475.1	1.3	4.5
pseudo-ii #Instances 41 total, 9 by all, 13 by at least one				
#solved	9	**13**	9	9
$\sum$CPU [sec]	>15,019.8	**1,697.1**	>15,054.8	>15,054.4
avg. NV	282,191.0	282,191.0	282,191.0	282,191.0
#CallsPOAC	-	0.0	0.0	0.0
mug #Instances 8 total, 4 by all, 5 by at least one				
#solved	4	**5**	4	4
$\sum$CPU [sec]	>3,600.1	**1,382.1**	>3,600.1	>3,600.1
avg. NV	94.0	94.0	94.0	94.0
#CallsPOAC	-	0.0	0.0	0.0
nengfa #Instances 12 total, 4 by all, 5 by at least one				
#solved	4	**5**	4	4
$\sum$CPU [sec]	>3,820.6	**2,592.7**	>3,822.5	>3,822.5
avg. NV	4,526.5	4,526.5	4,526.5	4,526.5
#CallsPOAC	-	0.0	0.0	0.0

are not triggering often, and thus does not significantly reduce the number of node visits and CPU time. Because the #CallsPOAC for NPOAC and $\cup_{cyc}$POAC are so small in comparison to POAC, we think that the reinforcement of PrePeak$^+$ likely needs to be modified in the context of weaker consistencies.

For the pseudo-ii, mug, and nengfa benchmarks, GAC solves the easy instances without triggering POAC (i.e., on all instances solved by GAC, the #CallsPOAC is 0). However, PrePeak$^+$ with POAC could solve addition instances that PrePeak$^+$ with NPOAC and $\cup_{cyc}$POAC could not solve.

Table 5.9 investigates a selection of benchmarks where PrePeak$^+$ with POAC is outperformed by GAC. On these benchmarks, we continue to see that the #CallsPOAC for PrePeak$^+$ with NPOAC and $\cup_{cyc}$POAC trigger is less than PrePeak$^+$ with POAC. Thus, for these benchmarks NPOAC and $\cup_{cyc}$POAC perform better than PrePeak$^+$ with POAC because their performance more closely matches that

Table 5.9: PrePeak$^+$ with POAC techniques on select benchmarks good for GAC

		PrePeak$^+$		
Algorithm	**GAC**	**POAC**	**NPOAC**	**$\cup_{cyc}$POAC**
rand-2-40-19 #Instances 50 total, 49 by all, 49 by at least one				
#solved	**49**	**49**	**49**	**49**
∑CPU [sec]	**47,461.4**	48,025.7	48,039.2	47,978.2
avg. NV	723,885.4	724,891.7	724,184.7	**723,548.6**
#CallsPOAC	-	40.3	29.1	26.6
tightness0.9 #Instances 100 total, 99 by all, 100 by at least one				
#solved	**100**	99	**100**	**100**
∑CPU [sec]	14,314.1	>15,098.2	**14,424.6**	14,524.9
avg. NV	19,984.6	**19,825.0**	19,971.9	19,778.5
#CallsPOAC	-	8.1	1.8	7.3
travellingSalesman-25 #Instances 15 total, 15 by all, 15 by at least one				
#solved	**15**	**15**	**15**	**15**
∑CPU [sec]	**4,307.4**	5,245.5	4,334.2	4,275.1
avg. NV	**192,353.4**	226,908.7	192,505.9	188,943.0
#CallsPOAC	-	33.7	2.9	4.9

of GAC.

We next test our hypothesis about the reward function of PrePeak$^+$ not being tuned properly for weaker consistencies, such as $\cup_{cyc}$POAC. To that end, adjust the reward for the common ratio r in PrePeak$^+$. Table 5.10 shows the result of changing the common-ratio powers from $r = (-1, 2, 3)$, the advocated method of Chapter 4, to $r = (-1, 0, 1)$. Changing r to $(-1, 0, 1)$ increases the number of calls to POAC from $r = (-1, 2, 3)$ (2,923.6 versus 31.8). However, despite calling HLC more frequently, the reduction in the number of node visits is relatively small. Thus, triggering more frequently cannot overcome that the consistency property is weaker.

Table 5.10: Changing the r reward for PrePeak$^+$ with $\cup_{cyc}$POAC

	$r = (-1, 2, 3)$	$r = (-1, 0, 1)$
#solved	**2,273**	2,248
$\sum$CPU [sec]	>**309,530.5**	>473,716.6
avg. NV	498,492.5	**471,911.0**
#CallsPOAC	31.8	2,923.6
#Instances 3,525 total, 2,245 by all, 2,276 by at least one		

Using neighborhoods or cycles with POAC weakens the resulting consistency property, which is not an effective strategy to do when triggering consistency.

We could potentially strengthen $\cup_{cyc}$POAC by computing the cycles dynamically during search rather than during pre-processing, however, it will likely still be too weak.

5.5 Cycles for Determining Singleton Tests

In the previous sections, the cycle basis was used to localize the AC filtering of each singleton test. In this section, we propose an alternative approach to instead of localize the singleton tests AC call to the cycles of which the singleton test is being conducted, we restrict the variables that we singleton test to those that appear in a cycle with the current instantiated variable. Empirically we find that this approach is not strong enough because instantiating the variable already breaks many cycles, and we need to break cycles in other areas of the CSP to get propagation. However, we report the idea and results as a lesson learned.

5.5.1 Determine Singleton Tests

Figure 5.9 shows a search progression. Variables x_1, x_2, and x_3 have been assigned by search values v_1, v_2, and v_3, respectively. The current variable being assigned by search is x_4 with value v_4. The future variables are $x_5 \ldots x_n$, which search is conducting lookahead over. Our typical lookahead is GAC, which will revise all the future variables. POAC operates by conducting a singleton test on every future variable and running GAC. We propose a hybrid, where we singleton test only the

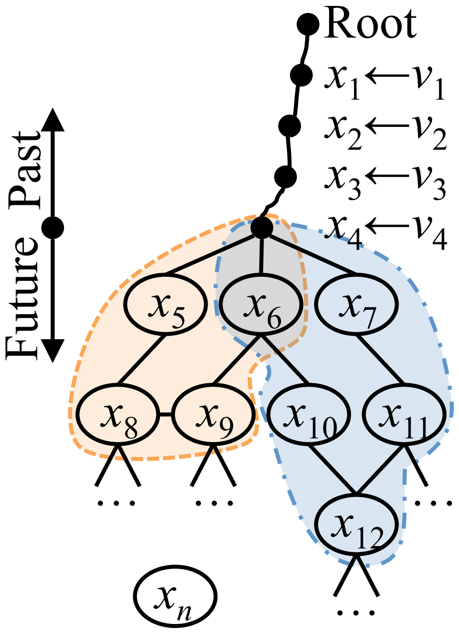


Figure 5.9: Search progression's past, current, and future variables

variables that appear in a cycle with the current variable (i.e., singleton test on variables $x_5 \ldots x_{12}$) and enforce GAC on the other variables (i.e., $x_{13} \ldots x_n$).

Using the approximation of computing the cycles of a variable, Section 5.3, allows the computation of cycles of the instantiated variable to be conducted dynamically during search. Thus, we compute the cycles in the graph induced by the current variable and the future variables (i.e., without the past variables) to determine which variables to singleton test in POAC.

5.5.2 Experimental Results

We set up our experiments the same as in Section 5.4.1. We compare STR2+ with APOAC and our new method of restricting singleton test variables of APOAC to those that appear in a cycle with the current variable, which we call APOAC$_{around\cup}$.

Table 5.11 shows the difference between APOAC and APOAC$_{around\cup}$. Overall, localized POAC does slightly better than APOAC in terms of completing more instances and CPU time. Of course, APOAC enforces the strongest consistency, and has the least number of node visits.

Table 5.11: Lookahead with GAC, APOAC, and Localized POAC

Algorithm	**GAC**	**APOAC**	**APOAC$_{around\cup}$**
#solved	2,277	2,132	2,151
$\sum$CPU [sec]	>371,758.3	>1,106,542.7	>1,069,967.5
avg. NV	481,013.5	**22,508.4**	269,870.1
#Instances 3,525 total, 2111 by all, 2296 by at least one			

Overall, the strategy does not work well because instantiating the variable already breaks the cycle. Thus, singleton testing only variables in the 'broken' cycles does not offer much additional filtering. Instead, we want to singleton test all variables to allow the broken cycles of the search-instantiated variable to propagate further.

Summary

In this chapter, we advocate the use of cycles to improve the performance of algorithms for enforcing POAC and provide empirical evidence of the benefit of our approach. Future work is to extend our cycles to other consistency algorithms, such as RNIC.

Chapter 6

Localizing Consistency to Triangles

Chapter 5 investigated where to enforce consistency using cycles of the CSP. In this chapter we focus on a special type of cycle: triangles. We are motivated by the consistency properties Partial Path Consistency (PPC), which operates on a triangulated primal graph of the CSP.

PPC is a staple for processing time in planning problems [Xu and Choueiry, 2003; Planken *et al.*, 2008] where its enforcement is able to solve the problem. For general CSPs, the enforcement of PPC has not widely been investigated and has only ever been evaluated as a pre-processing step to solving a CSP, largely due to its enforcement cost. In this chapter, we focus on the △PPC algorithm [Reeson, 2016] for enforcing PPC. We introduce new implementation improvements for △PPC and introduce various strengths of PPC by restricting its propagation queue. Finally, we combine PPC with our triggering techniques (Chapter 4) and show that it can be advantageous to enforce PPC during search.

Jégou [1993] introduces, for non-binary CSPs, a relational-consistency property, called hyper-3 consistency (H3C) that is 'symmetrical' to path consistency for binary CSPs. We extend H3C to a partial version, partial hyper-3 consistency (PH3C), which operates on triangles of a dual graph of the CSP, and propose the first algorithm for enforcing it based on the partial path consistency algorithm △PPC, which we call △PH3C. We theoretically and experimentally show that PH3C can solve the 'dubois' benchmark backtrack-free due to its structural configuration, which has never before been exploited.

6.1 Revisiting △PPC

Bliek and Sam-Haroud [1999] introduced BSH-PPC, the first algorithm for enforcing Partial Path Consistency (PPC) on binary CSPs. It operates on a queue of edges (i.e., constraints) that need to be tightened to make the problem PPC. The algorithm pops an edge $c_{i,j}$ from its queue and for each third variable k, such that the constraints $c_{i,k}$ and $c_{j,k}$ exists (i.e., a triangles of three variables), it tightens the constraint $c_{i,j}$ by joining and projecting these adjacent constraints, that is, $c_{i,j} \leftarrow c_{i,j} \cap \pi_{i,j}(c_{i,k} \bowtie c_{j,k})$

The state-of-the art algorithm for enforcing PPC is *Triangle Partial-Path Consistency* (△PPC) [Reeson, 2016]. △PPC improves on BSH-PPC in three ways:

1. The propagation queue is a queue of triangles, $\mathcal{Q}_t$ instead of a queue of edges.[1]
2. When a triangle is removed from the queue, Revise-Triangle revises all three edges at the same time.
3. Whenever an edge is updated, all of the triangles are pushed to the queue, except the one under consideration by the algorithm.

To ensure correctness, △PPC propagates its filtering of relations onto the articulation points and the cut edges in the graph.

---

[1] Processing triangles was originally exploited for Simple Temporal Problems in △STP [Xu and Choueiry, 2003] and P^{3}C [Planken *et al.*, 2008].

We give a detailed discussion of the △PPC algorithm, followed by our adaptation to using constraints represented as bit matrices, and using it to enforce different strengths of PPC.

6.1.1 The Algorithm

We follow Reeson's [2016] algorithm for △PPC with two modifications:

1. Allow △PPC to operates on any set of triangles, which are a set of three variables that are pairwise connected with a constraint in the CSP. Thus, the algorithm can be used with a subset of triangles, such as existing triangles in the primal graph for enforcing conservative path consistency, or to operate on some subset of the triangles, as we propose in Section 6.2.

2. Allow the propagation of the filtering by the algorithm to modify the entire CSP, rather than the articulation points and the cut edges of the graph. Our rationale for this change is that when PPC is combined with search, the domains of all variables need to be updated with respect to the updated relations, thus we interleave this operation in the call to △PPC.

We do not discuss all the technical aspects of △PPC, but instead focus on the relevant parts to understand these two changes.

Algorithm 13 reports our adaptation of the △PPC algorithm of Reeson [2016]. Our algorithm takes as input a set of triangles, *triangles*, which is a set of three variables that pairwise have a constraint between them. For enforcing PPC, *triangles* contains all of the triangles that appear in the triangulated CSP:

$$\textit{triangles} \leftarrow \{(x_k, x_j, x_i) \in \mathcal{V} | (i < j < k) \wedge (c_{i,j}, c_{i,k}, c_{j,k} \in \mathcal{C}')\}$$

Algorithm 13: $\triangle$PPC($\mathcal{P}$)

```
Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A binary CSP with a triangulated constraint graph;
       triangles: A set of triangles $\{(x_k, x_j, x_i)\}, \ldots\}$ to enforce on.
Output: Partially path consistent $\mathcal{P}$
1 $\mathcal{Q}_t \leftarrow$ triangles
2 ProjectAndSelect($\mathcal{C}$)
3 while $\mathcal{Q}_t \neq \emptyset$ do
4     $(x_k, x_j, x_i) \leftarrow$ Pop($\mathcal{Q}_t$)
5     $U \leftarrow$ Revise-Triangle($x_k, x_j, x_i$)
6     foreach edge $\in U$ do  $\mathcal{Q}_t \leftarrow \mathcal{Q}_t \cup TriangleEdge(edge) \setminus \{(x_k, x_j, x_i)\}$
7 return $\mathcal{P}$
```

Where $i < j < k$ is ordered by a perfect elimination ordering of the triangulated CSP and $\mathcal{C}'$ is the set of constraints in the triangulated CSP. $\triangle$PPC uses a propagation queue of triangles $\mathcal{Q}_t$ to determine which triangles need to be revised because of changes, which initially contains all triangles.

For correctness, $\triangle$PPC ensures that the articulation points and cut edges are kept updated with the latest relation filtering. Because of the combination with search, we instead update all variables and relations by projecting and selecting on all the constraints by calling ProjectAndSelect (Algorithm 14). ProjectAndSelect takes a set of constraints and checks if any domain values can be updated (i.e., projection of the constraints onto each variable in its scope) and propagates those changes on affected relations (i.e., selection on relevant constraints). A queue $\mathcal{Q}$ is used to process the constraints $c_{i,j}$ that have changed. The constraint is projected onto both x_i and x_j in its scope to see if any domain value can be removed. If x_i (or x_j) has been updated then all constraints that contain x_i (or x_j) are selected to ensure they contain only current domain elements and re-queued to further propagate any changes (Lines 8–12, and Lines 13–17, respectively).

After popping a triangle from the queue in $\triangle$PPC, we revise it by calling Revise-Triangle (Line 5 of Algorithm 13). We do not modify Revise-Triangle (Algo-

Algorithm 14: ProjectAndSelect(C)

```
Input: $C \subseteq \mathcal{C}$
Output: $U$: Set of updated constraints
$U \leftarrow \emptyset$
$\mathcal{Q} \leftarrow C$
while $\mathcal{Q} \neq \emptyset$ do
    $c_{i,j} \leftarrow$ Pop($\mathcal{Q}$)
    $U \leftarrow U \cup \{c_{i,j}\}$
    $origDom_i \leftarrow dom(x_i)$
    $origDom_j \leftarrow dom(x_j)$
    $dom(x_i) \leftarrow \pi_i(c_{i,j})$
    if $origDom_i \neq dom(x_i)$ then
        for $c' \in \mathcal{C}$ such that $(x_i \in scp(c')) \wedge (c' \neq c_{i,j})$ do
            $c' \leftarrow \sigma_{x_i \in dom(x_i)}(c')$
            $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{c'\}$
    $dom(x_j) \leftarrow \pi_j(c_{i,j})$
    if $origDom_j \neq dom(x_j)$ then
        for $c' \in \mathcal{C}$ such that $(x_j \in scp(c')) \wedge (c' \neq c_{i,j})$ do
            $c' \leftarrow \sigma_{x_j \in dom(x_j)}(c')$
            $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{c'\}$
return $U$
```

rithm 15), from that of Reeson's [2016], but discuss it here for completeness. Revise-Triangle calls Revise-3(i, j, k) (Algorithm 16 for each combination of i, j, k, which updates the relation $R_{i,j}$ using variable k. We differ our Revise-3 implementation from that of Reeson's [2016] in that we propagate the changes of the relations onto all variables, rather than articulation points in the graph. We accomplish this prop-

Algorithm 15: Revise-Triangle(i, j, k)

```
Input: $i, j, k \in \mathcal{X}$
Output: $U$: Set of updated constraints
$U \leftarrow \emptyset$
$U \leftarrow U \cup$ Revise-3($i, j, k$)
$U \leftarrow U \cup$ Revise-3($i, k, j$)
$U \leftarrow U \cup$ Revise-3($j, k, i$)
return $U$
```

Algorithm 16: Revise-3(i, j, k)

```
  Input: $i, j, k \in \mathcal{X}$
  Output: $U$: Set of updated constraints
1 $modified \leftarrow False$
2 foreach $(a, b) \in R_{i,j}$ do
3     if $\nexists c \in dom(k)$ such that $((a, c) \in R_{i,k}) \wedge ((b, c) \in R_{j,k})$ then
4         $R_{i,j} \leftarrow R_{i,j} \setminus \{(a, b)\}$
5         $modified \leftarrow True$
6 $U \leftarrow \emptyset$
7 if $modified$ then $U \leftarrow$ ProjectAndSelect($\{c_{i,j}\}$)
8 return $U$
```

agation by calling ProjectAndSelect (Line 7). Any relations that are modified by ProjectAndSelect, including $c_{i,j}$, are placed in the set of updated relations U for requeueing to $\mathcal{Q}_t$ for $\triangle$PPC (Line 6 of Algorithm 13).

6.1.2 Bit Implementation of the Constraints

Up to this point we represent a relation for a constraint by an enumerated table representing the constraint (i.e., a relation given in supports and extension). But, we propose an alternative representation of the constraints using a bit matrix. We call the version of $\triangle$PPC that utilizes a bit implementation of constraints $\triangle\text{PPC}^{bit}$.

In the bit representation of a constraint, we use a two-dimensional bit-matrix $M_{i,j}$ to represent each relation $R_{i,j}$ such that the location (a, b) is true iff the tuple (a, b) is allowed by $R_{i,j}$. In particular, we implement the $M_{i,j}$ as follows.

- The rows of the matrix represent a value in the domain of variable i and the columns represent a value in the domain of the variable j.
- A value at $M_{i,j}[a][b]$ is true iff $\langle a, b \rangle \in R_{i,j}$.
- $M_{i,j}[a]$ returns a bit vector of all of the values in j that support $i \leftarrow a$.

- We redundantly store the bit matrix M_{x_i,x_j} into M_{x_j,x_i} for ease of accessing the relation in both directions of x_i and x_j.
- For our implementation, we choose to implement $M_{i,j}$ as a vector indexed by the domain of i and $M[a]$ as a reversible sparse bit-set (RSparseBitSet) [Demeulenaere *et al.*, 2016]. A reversible sparse bit-set allows tuples to be restored in constant time when search backtracks.

REVISE-3^{bit} (Algorithm 17) is the updated pseudo-code of REVISE-3 which utilizes the bit presentation of the constraints in △PPC^{bit}. In Line 3, we obtain all the indices where $R_{i,j}[a]$ is true. This operation can be accomplished in constant time by counting the number of trailing 0s in the bit vector,[2] resulting in the index of the least significant bit. To find the next bit, the least significant bit is set to false and the process repeats itself until the bit vector is empty.

Algorithm 17: REVISE-3$^{bit}(i, j, k)$

```
  Input: $i, j, k \in \mathcal{X}$
  Output: $\mathcal{T}_d$ the set of removed tuples from $R_{i,j}$
1 $\mathcal{T}_d \leftarrow \emptyset$
2 foreach $a \in dom(i)$ do
3     foreach $b \in M_{i,j}[a]$ do
4         if $R_{i,k}[a] \& R_{j,k}[b] = false$ then
5             $R_{i,j}[a][b] \leftarrow false$
6             $\mathcal{T}_d \leftarrow \mathcal{T}_d \cup \{(a, b)\}$
7 return $\mathcal{T}_d$
```

6.1.3 Variations of PPC

Our PPC algorithm is based on first triangulating the graph of a binary CSP, perhaps by applying the MINFILL heuristic (Fig. 4.4 [Dechter, 2003b]). We organize the

---

[2] Counting the trailing 0s can be accomplished in constant time by using a CPU an instruction to count the number of trailing 0s, or if not available, a lookup table.

vertices along a Perfect Elimination Ordering (PEO). We denote as *elimination order* the traversal of the vertices from bottom to top and as *instantiation order* the traversal of the vertices from top to bottom (see Figure 6.1). We use the PEO to identify the sequence of triangles in the elimination order as the triangles (i, j, k) where $i < j < k$ sorted by first the largest value of k, then the largest value of j, then the largest value i as shown in Figure 6.2.

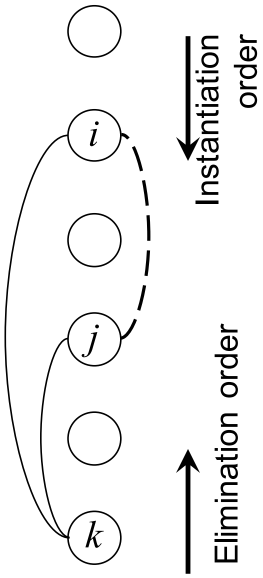


Figure 6.1: MinFill adds the edge (i, j) because of the existing edges (i, k) and (j, k)

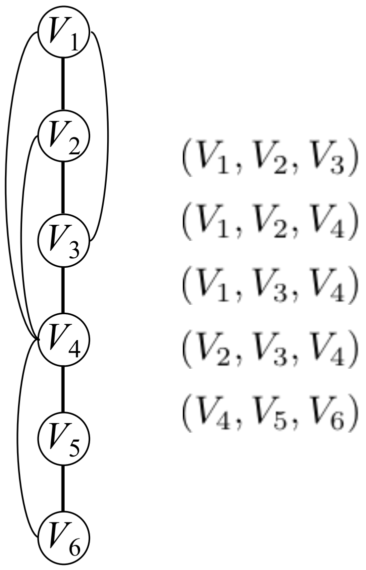


Figure 6.2: The sequence of triangles along the PEO of a triangulated graph

We investigate the following variations of PPC by restricting propagation in the identified triangles and limiting iteration over the propagation queue:

- Directional Path Consistency (DPC) [Dechter and Pearl, 1988] iterates through the triangles following the PEO along the elimination order (i.e., from bottom to top). DPC traverses the triangles only once and updates only the edge (i, j) in each triangle (i, j, k) such that $i < j < k$. Note that the edge (i, j) can be either an existing edge or is added by MinFill.

- Directional Partial Path Consistency (DPPC) iterates only once over the triangles following the PEO in the elimination order, updating all three edges of each triangle at each step.
- P^3C [Planken *et al.*, 2008] traverses twice the list of triangles: first in the elimination order then in the instantiation order. In the first traversal, it applies DPC, that is, updates the edge (i, j) in each triangle (i, j, k) such that $i < j < k$. Then, when traversing the triangles in the instantiation order, it updates the two edges (i, k) and (j, k).
- Two-swipes Directional Partial Path Consistency (2DPPC) applies DPPC twice, once following the elimination order and then following the instantiation order. (It goes through the sequence of triangles updating all three edges first following the elimination order then following the instantiation order.)
- PPC *repeatedly* iterates through the sequence of triangles up and down the PEO, starting from the elimination order and repeating until reaching a fixpoint. For each triangle, it updates all three edges.[3]

The Hasse diagram of the pruning effectiveness of the five listed algorithms is shown in Figure 6.3 with the weakest at the bottom and the strongest at the top. Notice that these queue strategies provide a natural 'how much' strategy for enforcing PPC. We refer the adaptation of △PPC by limiting the queue as: △PPC, △2DPPC, △DPPC, △P^3C, △DPC.

Reeson [2016] implements an ordered propagation queue for following the PEO ordering by processing all the triangles in a linear up and down ordering and using a 'flag' to determine if a triangle is in the propagation queue. Thus, processing the

---

[3] Reeson [2016] present this queue strategy as the algorithm σ-△PPC.

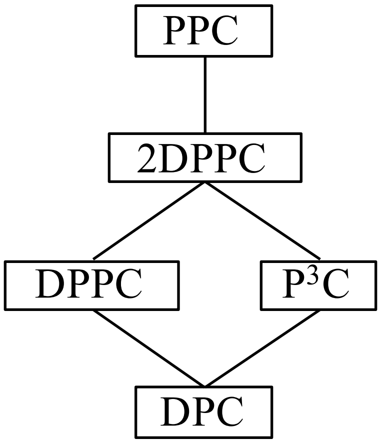


Figure 6.3: Pruning strengths of the proposed PPC-based consistencies

one direction of the queue involves $|triangles|$ checks of the flag, rather than iterating only through elements in the queue. We improve on this implementation to allow constant time access to the next element in the queue.

We continue to use a flag to determine if an element is present in the queue, but use two priority queues to iterate through only elements in the:

1. a forward queue where the priority of a triangle is its position in the PEO,
2. and a backward queue where the priority of a triangle is its position from the end of the PEO.

We use a flag to indicate what queue to pop elements from, initially indicating that the forward queue is to be used. Elements are popped from the flagged direction's queue until it is empty, at which point the flag changes, indicating that the other queue is to be popped from. This process continues until both queues are empty.

When inserting an element the priority of the element is compared with the priority of the last popped element to determine which queues it is inserted into using the ordering of the currently-flagged queue. If the element to be inserted has a priority after the previously element, its added to the flagged queue, otherwise it is added to the non-flagged queue.

Maintaining two priority queue is beneficial when the queue has few elements in it. Rather than checking a flag on all the triangles, the queue can be accessed directly.

6.2 Generating Triangulated Edge Constraints

Partial Path Consistency (PPC) operates on triangles of the triangulated primal graph, generating new constraints for the triangulated edges. Although generating the triangulated edges is less effort than enforcing on a complete graph (i.e., Path Consistency), generating additional edges may not always be worthwhile. Indeed, generating these new edges can be costly in terms of CPU time and memory. In this section we propose a method for selecting a subset of triangles to operate on, and thus, a subset of triangulated edges to generate. Operating on a subset of triangles introduces a new consistency properly strictly stronger than CPC but strictly weaker than PPC. In particular, we advocate utilizing a tree decomposition of the CSP to determine the subset triangles.

6.2.1 Using the Separators of a Tree Decomposition

The separators between clusters identifies the variables that propagate changes from one cluster to another. We propose to identify only triangles that have a certain number of variables in the separator. Three strategies can be derived:

1. The triangle has at least one variable in the separator, corresponding to selecting triangles that at least ‘touches’ the separator.

2. The triangle has at least two variables in the separator, corresponding to selecting triangles that have an edge in the separator.
3. The triangle has all three variables in the separator, corresponding to selecting triangles completely contained in the separator.

Using the separators in this manner does not allow us to easily discriminate which triangles should be selected in the case that many of the triangles appear in separators. In the next section, we instead investigate looking at where the triangles appear in the clusters of a tree decomposition.

6.2.2 Using the Clusters of a Tree Decomposition

Triangles that appear in many clusters allow it to communicate changes across the CSP easily. Thus, we enumerate all of the triangles of a graph (e.g., the triangulated primal graph in the case of PPC) counting how many clusters of the tree decomposition the triangle appears in.[4] We start selecting triangles, which may be either an existing triangle or contains a triangulated edge, starting with the triangles that appear in the most number of clusters. We select some threshold $\theta_{\triangle}$ of the number of triangles to accept. We include all the triangles that appear in the same number of clusters as the ${\theta_{\triangle}}^{th}$ triangle to account for ties. After this point, we add all existing triangles in the original, un-triangulated, graph, some of which may have already been

Example 6 Figure 6.4 the primal graph of an example CSP. Figure 6.5 shows the

---

[4] The number of separators the triangle appears in is always one fewer than the number of clusters a triangle appears in. However, we argue that the clusters provides a more accurate measure given that it can distinguish between triangles that appear in one cluster (i.e., in no separators) and those that appear no clusters.

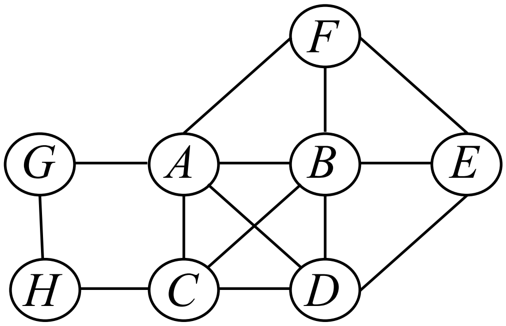


Figure 6.4: The primal graph

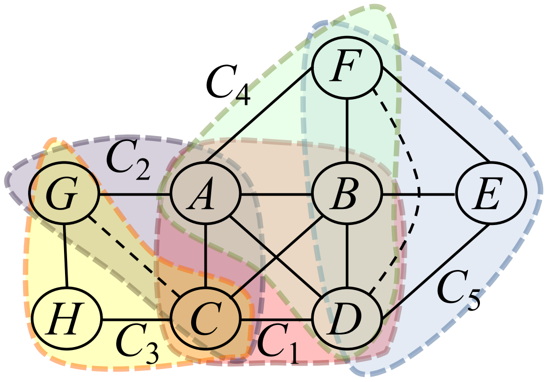


Figure 6.5: Triangulated primal graph and its maximal cliques

triangulated primal graph and its maximal cliques. Figure 6.6 shows a tree decomposition for this example. There are twelve triangles in the triangulated primal graph:

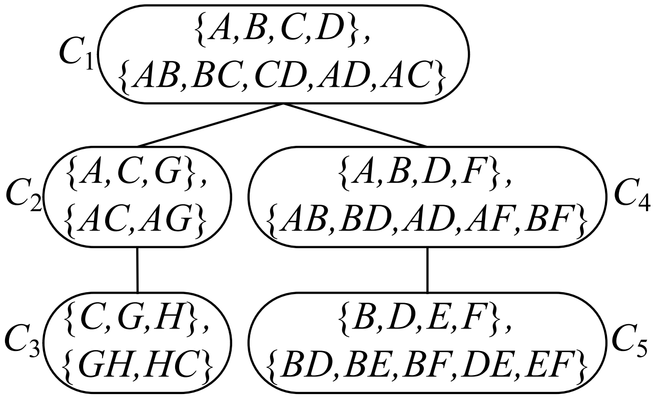


Figure 6.6: A tree decomposition of the CSP in Figure 6.4

(A, B, C), (A, B, D), (A, B, F), (A, C, D), (A, C, G), (A, D, F), (B, C, D), (B, D, E), (B, D, F), (B, E, F), (C, G, H), (D, E, F). The triangles (A, B, D) and (B, D, F) appears in two cliques of the tree decomposition, while the other triangles appear in one clique. For $\theta_\triangle = 1$ or 2 the triangles (A, B, D) and (D, B, F) are accepted. For $\theta_\triangle \geq 3$ all twelve triangles are accepted.

In practice, we set $\theta_\triangle = 2n$, where n is the number of variables. We justify this choice of $\theta_\triangle$ because it allows each variable to have two triangles associated with it, either containing triangulated or original constraints.

6.2.3 Implementing Triangle Generation

The constraint definition of a triangulated edges in the primal graph is a universal constraints (i.e., allowing all combinations of values between the two variables). Enforcing PPC may tighten these constraint. By generating the constraints through a single operation of PPC, that is, by joining two adjacent relations and projecting onto the scope, reduces both the memory consumption of the program and the time to iterate through the resulting relations. Following a perfect elimination ordering (PEO) allows us to generate the relations in just this fashion. The definition of PEO ensures that two adjacent relations will already exist, or have already been generated, when doing this operation.

Algorithm 18 generate the triangulated edges by joining and projection existing edges following a perfect elimination ordering. The join of relations $R_{i,k}$ and $R_{k,j}$ in Line 8 will be either existing or previously generated relations because of the definition of a perfect elimination order.

If a subset of triangles are to be generated, as discussed in Sections 6.2.1 and 6.2.2, not all the triangulated edges may need to be generated. In such a situation generating a new constraint in Line 8 of Algorithm 18 may require the join with a non-generated relation, and thus, the resulting generated constraint is a universal constraints. In practice we find that generating these universal constraints is wasteful in memory consumption and that, for some problem instances, generating the selected universal constraints consumes *more* memory than generating all the triangulated edges following Algorithm 18 and discarding the not required constraints at the end of the

```
Algorithm 18: GenerateEdges($\mathcal{P}$, $peo$)
  Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A binary CSP; $peo = (|\mathcal{V}|, \ldots, 1)$: a perfect
         elimination ordering on $\mathcal{V}$
  Output: A triangulated CSP $\mathcal{P}' = (\mathcal{V}, \mathcal{D}, \mathcal{C}')$ and a set of triangles $triangles$
  $\mathcal{C}' \leftarrow \mathcal{C}$
  $triangles \leftarrow \emptyset$
  for $k \leftarrow |\mathcal{V}|$ down to $3$ by $-1$ do
      for $j \leftarrow k - 1$ down to $2$ by $-1$ do
          for $i \leftarrow j - 1$ down to $1$ by $-1$ do
              if $C_{i,k} \in \mathcal{C}'$ and $C_{k,j} \in \mathcal{C}'$ then
                  if $C_{i,j} \notin \mathcal{C}'$ then
                      $R_{i,j} \leftarrow \pi_{i,j}(R_{i,k} \bowtie dom(x_k) \bowtie R_{k,j})$
                      $\mathcal{C}' \leftarrow C_{i,j}$
                  $triangles \leftarrow triangles \cup \{(x_k, x_j, x_i)\}$
  return $((\mathcal{V}, \mathcal{D}, \mathcal{C}'), triangles)$
```

operation.

Algorithm 19 avoids such situations by determining all the temporary and permanent edges to be generated for a given subset of triangles. The algorithm operates by recording the triangles that create a triangulated edge following the PEO ordering. The three for loops of Lines 5–7 finds when the edges are first found (i.e., generated) and stores them in *edgeGeneratedFrom*, which is a stack of edges generated. The for loop of Line 13 goes in the reverse order of generated edges determining if the generated edge appears in a triangle in Line 15, needs to be temporarily generated (i.e., the edge will be used for generating another edge) in Line 17, or if the edge does not need to be generated (ignored). The for loop of Line 19 goes through all of *edgeGeneratedFrom* in the order of generation, generating edges if they are either required in Line 22 or required temporarily in Line 25. Finally, in Line 27 all of the temporary edges are removed.

Algorithm 19: GENERATESOMEEDGES($\mathcal{P}$, peo, tri)

```
Input: $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$: A binary CSP; $peo = (|\mathcal{V}|, \ldots, 1)$: a perfect
       elimination ordering on $\mathcal{V}$; $tri$: An set of triangles $\{\{k, j, i\}, \ldots\}$ to
       be generated where $i < j < k$ in $peo$
Output: $\mathcal{P}' = (\mathcal{V}, \mathcal{D}, \mathcal{C}')$
$\mathcal{C}' \leftarrow \mathcal{C}$
$\mathcal{C}_{temp} \leftarrow \emptyset$
$foundEdges = \emptyset$
$edgeGeneratedFrom = [\ ]$
for $k \leftarrow |\mathcal{V}|$ down to 3 by $-1$ do
    for $j \leftarrow k - 1$ down to 2 by $-1$ do
        for $i \leftarrow j - 1$ down to 1 by $-1$ do
            if $C_{i,k} \in \mathcal{C}'$ and $C_{k,j} \in \mathcal{C}'$ and $C_{i,j} \notin foundEdges$ and $C_{i,j} \notin \mathcal{C}'$
             then
                $PushBack((k, j, i), edgeGeneratedFrom)$
                $foundEdges \leftarrow foundEdges \cup \{i, j\}$
$requiredEdges = \emptyset$
$requiredEdgesTemp = \emptyset$
for $t \leftarrow |edgeGeneratedFrom|$ down to 1 by $-1$ do
    $(k, j, i) \leftarrow edgeGeneratedFrom[t]$
    if $\{k, j, i\} \in tri$ then
        $requiedEdges \leftarrow requiredEdges \cup \{(i, j), (i, k), (j, k)\}$
    else if $(i, j) \in requiedEdges$ or $(i, j) \in requiedEdgesTemp$ then
        $requiredEdgesTemp \leftarrow requiredEdgesTemp \cup \{(i, j), (i, k), (j, k)\}$
for $t \leftarrow 1$ upto $|edgeGeneratedFrom|$ do
    $(k, j, i) \leftarrow edgeGeneratedFrom[t]$
    if $(i, j) \in requiedEdges$ then
        $R_{i,j} \leftarrow \pi_{i,j}(R_{i,k} \bowtie dom(k) \bowtie R_{k,j})$
        $\mathcal{C}' \leftarrow C_{i,j}$
    else if $(i, j) \in requiedEdgesTemp$ then
        $R_{i,j} \leftarrow \pi_{i,j}(R_{i,k} \bowtie dom(x_k) \bowtie R_{k,j})$
        $\mathcal{C}_{temp} \leftarrow C_{i,j}$
$C_{temp} \leftarrow \emptyset$
return $(\mathcal{V}, \mathcal{D}, \mathcal{C}')$
```

6.2.4 Decision Tree for Selecting Triangles for PC

On some problem instances, it is ill-advised to triangulate the graph and generate new constraints. On the one end of the spectrum, the constraint graph is so dense that AC guarantees ‘quick’ propagation and is thus sufficient for search. In less dense graphs, it is sufficient to run a PPC algorithm on only the existing triangles in the graph. On the other hand of the spectrum, we do need to consider triangles that are formed as a result of triangulating the graph.

We propose the selection policy shown in Figure 6.7 to determine which triangles our PPC-based algorithms should operate on given the density d_p of the primal graph.[5] The goal of this deliberation is to adjust the strength of PPC to the topology of the primal graph. Paraphrasing the content of Figure 6.7:

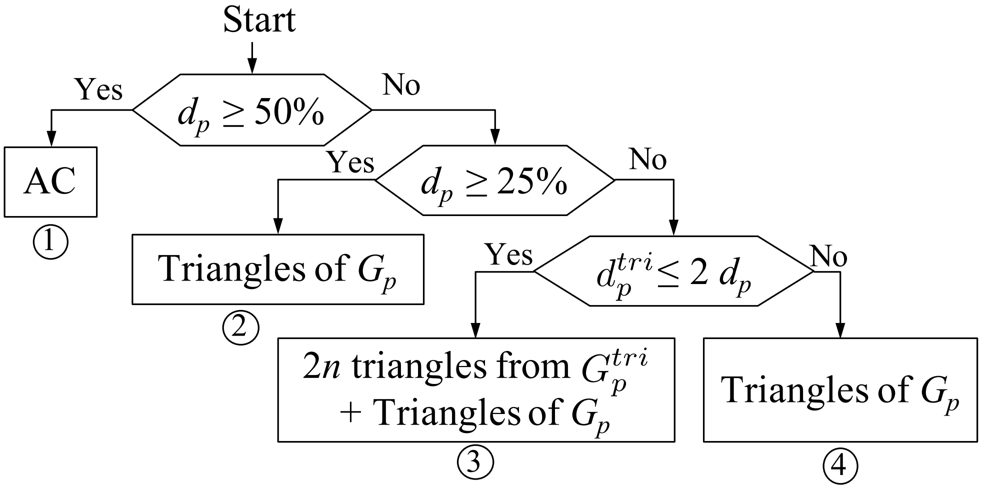


Figure 6.7: Selecting the triangles for PPC

- We consider that, at a density of the primal graph of 50% or more (i.e., $d_p \geq 50\%$), AC fully propagates the impact of a variable instantiation. HLC typically

---

[5] This decision tree is similar to the one we advocated for RNIC [Woodward *et al.*, 2011b].

yields only overhead but no further filtering. For this reason, we choose to simply enforce AC (see leaf 1 in Figure 6.7).

- In the remaining cases, we choose to *always* exploit the existing triangles from the graph and sometimes we maybe need to add a few more triangles.

- If the density of the primal graph is greater than 25% (i.e., $d_p \geq 25\%$), there is enough communication in the primal graph, and we choose to exploit *only* existing triangles from the graph (i.e., Triangles of G_p in leaf 2 in Figure 6.7).

- Otherwise, we examine the triangulated primal graph.

- If triangulated primal graph G_p^{tri} more than doubles the number of edges (i.e. $d_p^{tri} > 2d_p$), then the number of the additional triangles resulting from triangulation can be overwhelming and may cause a serious overhead. We estimate that the existing triangles are numerous enough to 'carry the propagation' over the primal graph. For this reason, we choose to operate only on the existing triangles in the original primal graph (see leaf 4 in Figure 6.7).

- When primal-graph triangulation does not prohibitively add to the density of the primal graph (i.e., $d_p^{tri} \leq 2d_p$), then we estimate that the triangulated primal graph is not too dense and that the advantage of boosting propagation outweighs the overhead of increasing the number of triangles to process. In this case,

    1. We choose first the $2n$ 'most critical' triangles from the triangulated primal graph. By critical, we mean those triangles that appear in the largest number of clusters in some tree decomposition of the triangulated primal graph (i.e, $2n$ triangles of G_p^{tri} in leaf 3 of Figure 6.7). The chosen triangles

may either be a triangle from the original primal graph or contain triangulated edges. Further, we include all the triangles that appear in the same number of clusters as the $2n^{th}$ triangle (i.e., we include all ties).

2. After this point, we add all existing triangles in the original primal graph unless they are already added.

The decision is illustrated as leaf 3 in Figure 6.7.

6.2.5 Watching Memory Usage

Although our proposed strategy attempts to reduce the number of triangles generated (Section 6.2.4) and avoids generating universal constraints (Section 6.2.3), generating additional constraints may still cause the program to go over its memory limit. For this reason, we watch the memory usage as our code program is generating *each* new constraint. If the memory usage is within 1GB of our threshold,[6] we stop generating new constraints, remove any added constraints, and default to only using existing triangles (i.e., leaf 4 in Figure 6.7).

To accomplish this, we add a check after Line 20 of Algorithm 19 (Section 6.2.3) to compare the current memory usage with our memory limit. If it is over our threshold, we terminate the algorithm after clearing all generated constraints.

Importantly, this mechanism for watching memory usage when generating new constraints is general. It is orthogonal and applicable beyond the usage of a decision tree.

---

[6] In our experiments, we limit the memory usage to 8GB.

6.3 Experimental Evaluation of △PPC

In this section, we evaluate the effectiveness of △PPC, *which has never before been evaluated during search.* To this end, we consider the problem of finding a single solution to a CSP using backtrack search, the dom/wdeg variable ordering heuristic [Boussemart *et al.*, 2004], and real-full lookahead [Haralick and Elliott, 1980].

We first discuss our experimental setup. We then validate our approach in four directions:

1. We evaluate the different variations of PPC proposed in Section 6.1.3, showing that the variation of P^3C performs best. The remaining directions are thus conducted over the P^3C variant.
2. We demonstrate the good performance of our decision tree for selecting the triangles to use, Section 6.2.4, during pre-processing.
3. We next show the performance of using the decision tree for selecting the triangles as real-full lookahead. Again, we insist that PPC-based algorithms have never before been evaluated during search.
4. We compare triggering △PPC using the strategies PrePeak, BTWatch, and PP-BTWatch of Chapter 4.

6.3.1 Experimental Setup

We set up our experiments as follows. We use GAC2001 [Bessière *et al.*, 2005] as the GAC algorithm, which is always maintained during search. We use GAC2001 instead of STR2+ as we find that it performs better than STR algorithms on binary CSPs.[7]

---

[7] Table E.2 in Appendix E compares the search performance using GAC2001 and STR2+ on binary CSPs.

We use the △PPC algorithm for enforcing PPC using the directional queue. The variations of PPC provide a 'how much' strategy for terminating the PPC early, thus we do not use the how much strategy of Section 4.2. We compare the performance of △PPC with that of sCDC1 [Lecoutre *et al.*, 2007], which enforces a stronger property than PPC and does not require generating new constraints.

For all ordering heuristics, we do not allow the constraints added by MINFILL to change the degree of a variable. Further, we do not allow AC to operate on the added constraints and use them for further propagation. We deliberately choose to prevent such interaction between AC and our PPC algorithms in order to more precisely assess the actual impact of PPC.

We use the benchmark problems available from Lecoutre's website,[8] including all binary benchmarks with at least one instance with a primal graph of density less than 50%, resulting in 50 benchmarks with 2,622 instances used in our experiments.[9] We use a time limit of 60 minutes per instance and 8GB of memory.

In the tables that summarize our results, we report for each algorithm, where applicable:

- The number of instances solved in a given benchmark (#solved).
- The number of node visits averaged over the instances completed by all algorithms (avg. NV).
- The sum of the CPU time in seconds of the run time of an algorithm for all the instances in a benchmark completed by any of the compared algorithms ($\sum$CPU). When an algorithm does not terminate within the allocated time, we

---

[8] `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

[9] Table E.1 in Appendix E list the selected benchmarks.

add 3,600 seconds to the CPU time and indicate with a '>' sign that the time reported is a lower bound.

- The average number of calls to HLC over the instances completed by all algorithms (#CallsHLC). GAC does not call a HLC, thus the number of calls to HLC is reported as '-'.
- Finally, we highlight, with a boldface, the best value in a given row.

6.3.2 Comparison of Variations of PPC

We study the cost of enforcing the five variations of PPC introduced in Section 6.1.3. We compare the performance on dom/deg, shown in Table 6.1, and dom/wdeg, shown in Table 6.2.

Table 6.1: △PPC variants as RFL with dom/deg

Algorithm	**AC**	**△DPC**	**△DPPC**	**△P^{3}C**	**△2DPPC**	**△PPC**
#Solved	1,904	1,785	1,789	1,814	1,777	1,711
#Memout	118	232	232	232	232	232
$\sum$CPU [sec]	>649,176.2	>1,724,091.6	>1,727,163.7	>1,576,772.6	>1,835,310.4	>2,111,836.4
Avg. #NV	238,828.6	15,072.8	15,072.8	3,395.6	2,099.9	1,406.8
#Instances 2,622 total, 1,582 by all, 2,015 by at least one						

Table 6.2: △PPC variants as RFL on dom/wdeg

Algorithm	**AC**	**△DPC**	**△DPPC**	**△P^{3}C**	**△2DPPC**	**△PPC**
#Solved	2,079	1,831	1,794	1,824	1,707	1,774
#Memout	118	232	232	232	232	231
$\sum$CPU [sec]	>184,865.7	>1,789,327.5	>1,982,639.4	>1,790,378.9	>2,377,861.5	>2,114,243.0
Avg. #NV	22,296.0	6,660.7	6,660.7	2,653.8	1,152.7	1,780.1
#Instances 2,622 total, 1,686 by all, 2,087 by at least one						

For both ordering heuristics, AC performs better than any of the PPC variations. This result is predictable because PPC is likely to be too strong for most problem instances, which we address in our experiments in Section 6.3.5. The statistical rankings, according to a paired t-test, of the considered variations are as follows:

$$\text{dom/deg: } \triangle\text{P}^3\text{C} \succ \triangle\text{DPC} \sim \triangle\text{DPPC} \succ \triangle\text{2DPPC} \succ \triangle\text{PPC}$$
$$\text{dom/wdeg: } \triangle\text{P}^3\text{C} \sim \triangle\text{DPC} \succ \triangle\text{DPPC} \succ \triangle\text{2DPPC} \succ \triangle\text{PPC}$$

where $A \succ B$ denotes that A is statistically better than B and $A \sim B$ denotes that there are not statistically distinguishable. Because $\triangle$P^{3}C performs statistically the best in both ordering heuristics, we evaluate using the $\triangle$P^{3}C variant in the remainder of this section.

6.3.3 As Pre-Processing

As a first step towards evaluating $\triangle$P^{3}C, we evaluate enforcing HLC at pre-processing while running GAC as RFL. We evaluate $\triangle$P^{3}C with of our three strategies for selecting triangles: Existing, $2n$ New with all existing triangles (Section 6.2.2), and using the Decision Tree (Figure 6.2.4). Table 6.3 gives the overall performance on all the benchmarks evaluated. All $\triangle$P^{3}C techniques solves more instances than GAC.

Table 6.3: $\triangle$P^{3}C on subsets of triangles at pre-processing followed by GAC as RFL

		$\triangle$P^{3}C			
	GAC	**Existing**	**$2n$ New**	**DT**	**sCDC1**
# solved	2,081	2,082	2,085	**2,089**	2,064
$\sum$CPU [sec]	>282,490.0	>290,102.2	>338,120.1	>**264,855.9**	>426,359.5
avg. NV	129,357.1	126,535.6	147,326.7	126,711.1	124,306.6
#Instances 2,622 total, 2,031 by all, 2,116 by at least one					

However, ‘Existing’ and ‘$2n$ New’ both have a larger CPU time than GAC. This illustrates that although PPC can be helpful to solve more instances, there is an overhead associated with it that can be detrimental to CPU time. The decision tree (DT) helps overcome some of the CPU time overhead. Although the strongest in terms of filtering, sCDC1 solves the fewest number of instances and takes the largest CPU time.

6.3.4 As Real-Full Lookahead

We now evaluate $\triangle$P^{3}C as real-full lookahead, *which has never been evaluated before.* Table 6.4 gives the overall performance on all the benchmarks evaluated. When used as real-full lookahead, the sCDC1 and the $\triangle$P^{3}C techniques solve a *fewer* number of instances compared to the running at pre-processing only (Table 6.3 of Section 6.3.3). Solving fewer instances is not surprising as these algorithms can be expensive to enforce. This result illustrates the necessity of evaluating with a ‘when’ technique (e.g., PrePeak), as we evaluate in Section 6.3.5.

Table 6.4: The performance of enforcing consistency as RFL

		$\triangle$P^{3}C			
	GAC	**Existing**	**$2n$ New**	**DT**	**sCDC1**
# solved	**2,081**	1,951	1,915	1,988	1,790
$\sum$CPU [sec]	>**250,090.0**	>999,926.2	>1,399,620.4	>765,926.3	>1,779,505.9
avg. NV	47,777.2	20,672.2	22,786.8	28,770.9	**1,456.6**
#Instances 2,622 total, 1,756 by all, 2,107 by at least one					

sCDC1 has the fewest number of node visits, which is expected as it enforces a stronger consistency than the $\triangle$P^{3}C techniques. Among the $\triangle$P^{3}C techniques, the decision tree solves the most number of instances, showing it makes a good compromise between existing triangles and new triangles.

In Table 6.5 we highlight the good performance of enforcing $\triangle$P^{3}C as RFL on certain benchmarks. These benchmarks show the promise of enforcing $\triangle$P^{3}C during search. For the mug benchmark, selecting $2n$ triangles allows $\triangle$P^{3}C to solve the instance backtrack-tree. For the jobShop (e0ddr1 and enddr1) and super-os-taillard-4 benchmarks, the decision tree was able to capture the correct set of triangles to use,

Table 6.5: The good performance of $\triangle$P^{3}C as RFL on select benchmarks

	GAC	**$\triangle$P^{3}C**			**sCDC1**
		Existing	**$2n$ New**	**DT**	
mug #Instances 8 total, 4 by all, 8 by at least one					
# solved	4	5	**8**	**8**	**8**
$\sum$CPU	>14,400.1	>11,549.1	**100.3**	**100.3**	160.8
jobShop-e0ddr1 #Instances 10 total, 4 by all, 7 by at least one					
# solved	5	4	**7**	**7**	4
$\sum$CPU	>7,563.3	>11,248.3	1,266.0	**1,210.4**	>16,763.0
jobShop-enddr1 #Instances 10 total, 5 by all, 10 by at least one					
# solved	9	9	9	**10**	6
$\sum$CPU	>3,750.4	>5,958.8	>5,397.5	**1,860.3**	>25,122.7
super-os-taillard-4 #Instances 30 total, 20 by all, 30 by at least one					
# solved	28	**30**	**30**	**30**	20
$\sum$CPU	>9,151.1	7,959.4	14,167.1	**7,956.4**	>46,437.9
QWH-20 #Instances 10 total, 9 by all, 10 by at least one					
# solved	9	**10**	**10**	**10**	9
$\sum$CPU	>4,892.5	**2,838.3**	3,388.3	2,879.4	>5,603.9

solving more instances than GAC and sCDC1. For the QWH-20 benchmark, all the $\triangle$P^{3}C techniques solved 30 instances. The decision tree correctly selects the existing triangles.

Because the decision tree performs the best, we evaluate using the decision tree in the remainder of this section, which we denote as $\triangle$P^{3}C$^+$.

6.3.5 Triggering PPC

As seen in the previous section, $\triangle$P^{3}C$^+$ is too costly to run as RFL in general. We combine $\triangle$P^{3}C$^+$ with the three strategies of Chapter 4, BTWatch, PP-BTWatch, and PrePeak, to enforce these $\triangle$P^{3}C$^+$ selectively (i.e., when).

Table 6.6 gives the overall performance on all the benchmarks evaluated. All three triggering strategies solve more instances than GAC in smaller CPU time. BTWatch performs the best, solving the most number of instances in the smallest CPU time.

Table 6.6: The performance of triggering $\triangle P^3C^+$

Algorithm	**GAC**	**BTWatch**	**PP-BTWatch**	**PrePeak**
# solved	2,081	**2,091**	2,088	2,089
∑CPU [sec]	>214,090.0	>**192,656.5**	>195,373.9	>197,682.7
avg. NV	137,128.4	132,142.1	132,111.2	132,105.1
#CallsHLC	-	7.6	6.2	10.9
#Instances 2,622 total, 2,073 by all, 2,097 by at least one				

Figure 6.8 shows the cumulative number of instances completed by GAC, and using BTWatch, PP-BTWatch, and PrePeak with PPC as time increases. From the graph, the three triggering techniques dominate GAC. The triggering techniques are equivalent.

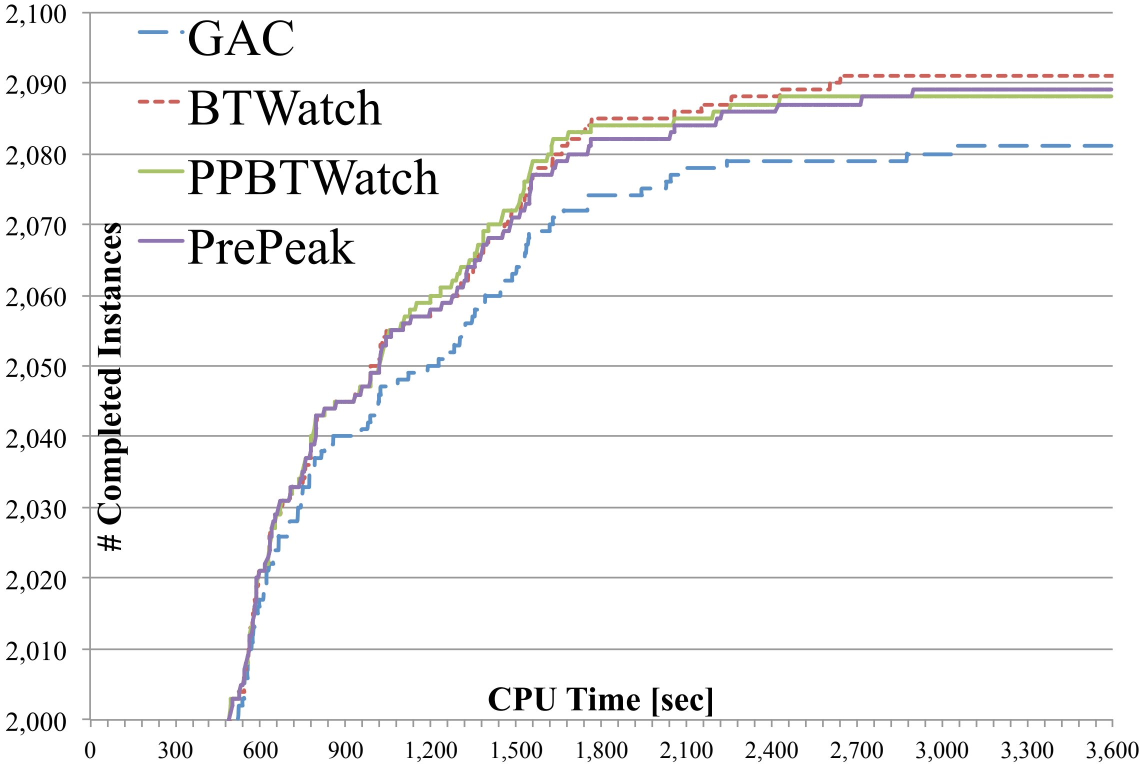


Figure 6.8: Cumulative instances completed by CPU time for triggering $\triangle P^3C^+$

6.4 Hyper-3 Consistency

We extend the existing algorithms for CPC and PPC to Conservative and Partial Hyper-3 Consistency.

Path consistency (also known as 3-consistency) ensures that every two variables can be consistently extended to a third. Hyper-3 Consistency ensures that every two *relations* can be consistently extended to a third.

Definition 20 Hyper-3 consistent [Jégou, 1993]: *Let* π *and* $\bowtie$ *be a relation projection and join, respectively. A CSP* $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ *is hyper-3 consistent iff* $\forall c_i, rel(c_i) \neq \emptyset$, $\forall c_j, c_k \in \mathcal{C}$, $\pi_I(rel(c_j) \bowtie rel(c_k)) \subseteq \pi_I rel(c_i)$, *where* $I = (scp(c_j) \cup scp(c_k)) \cap scp(c_i)$.

No algorithm exists for enforcing hyper-3 consistency. In this section, we extend the definition of hyper-3 consistency to partial hyper-3 consistency (PH3C) and give an algorithm for enforcing it.

6.4.1 Extending Hyper-3 Consistency

Conservative Path Consistency (CPC) [Debruyne, 1999] and Partial Path Consistency (PPC) [Bliek and Sam-Haroud, 1999] can easily be extended to Conservative Hyper-3 Consistency (CH3C) and Partial Hyper-3 Consistency (PH3C), respectively by looking at the dual graph of the CSP. CH3C operates on the dual graph of the problem and PH3C operates on the triangulated dual graph of the problem. Like CPC and PPC, CH3C and PH3C are equivalent when the dual graph is triangulated.

We theoretically compare H3C and its variants to that of Path Consistency (PC) on binary CSPs.

Theorem 14 *On binary CSPs, Hyper-3 Consistency* (H3C) *is strictly stronger than Path Consistency* (PC).

Proof: PC extends every two variables to a third. H3C extends every two tuples $\tau_i \in c_i$ and $\tau_j \in c_j$ to a third tuple $\tau_k \in c_k$. On binary CSPs $|scp(c_i) \cap scp(c_j)| \leq 1$ and τ_i and τ_j involve either three or four variables. Thus, on binary CSPs H3C extends every three (and four) variables to τ_k, which is clearly stronger than PC. $\square$

Further, the conservative version of H3C and PC (i.e., enforce on existing triangles) Conservative Hyper-3 Consistency (CH3C) is strictly stronger than Conservative Path Consistency (CPC). This follows from Theorem 14.

On binary CSPs, PH3C and PPC are incomparable because of the possible differences possible triangulations on the dual and primal graph. Notice, that the triangulations of the dual and primal graph will rarely result in the same set of triangles because of the differences in the structures of the graphs. However, assuming the same triangles are generated for PH3C and PPC (i.e., triangulating both graphs will result in the same set of triangles), PH3C is strictly stronger than PPC, which follows from how CH3C is strictly stronger than CPC.

The tree width of a tree decomposition can be used to characterize the complexity of the CSP [Freuder, 1982].

Theorem 15 *If the primal graph of a binary CSP $\mathcal{P}$ has a tree decomposition where every cluster is a clique of size at most three (i.e., a triangle) and $\mathcal{P}$ is Strong Partial-Path Consistency* (sPPC)*, then the domains of $\mathcal{P}$ are minimal.*

Proof: Re-phrasing of Theorem 4.5 of [Dechter, 2003a][10] to fit this framework. $\square$

Theorem 16 *If the dual graph of a CSP $\mathcal{P}$ has a tree decomposition where every cluster is a clique of size at most three (i.e., a triangle) and is Partial Hyper-3 Consistent then the relations of $\mathcal{P}$ are minimal.*

---

[10] Also a re-phrasing of Theorem 2.4 of [Dechter and Pearl, 1988] and Theorem 1 of [Freuder, 1982].

Proof: Follows from Theorem 15. ☐

In the experiments section, we exploit Theorem 16 to illustrate the theoretical rational why the 'dubois' benchmark is tractable.

6.4.2 Extending △PPC to △PH3C

Given how similar the properties of CPC and PPC are to CH3P and PH3P, we can use the same △PPC algorithm to enforce CH3P and PH3P. The only difference is the input triangles, which are generated from the dual graph rather than the primal graph. Recall that in the dual graph constraints are equality constraints. To enforce CH3P/PH3P on the problem, we have to enumerate the equality constraints to disallow combinations of two tuples.

Similar to the variations in the strengths of PPC in Section 6.1.3, we can apply the same restrictions of the propagation queue to obtain different strengths of PH3C:

- Directional Hyper-3 Consistency (DH3C), iterating through the triangles in the same fashion as DPC.
- Directional Partial Hyper-3 Consistency (DPH3C) iterates in the same fashion as DPPC.
- P^3H3C iterates through the triangles in the same fashion as P^3C.
- Two-swipes Directional Partial Hyper-3 Consistency (2DH3CC) iterates through the triangles in the same fashion as 2DPPC.
- PH3C iterates through all the triangles until reaching a fixpoint.

6.4.3 Bit Implementation for $\triangle$PH3C

Empirical evaluations of $\triangle$PH3C shows that enumerating the equality constraints requires too much memory to store and too much time to be useful in practice. However, choosing a different representation of the constraints allows us to solve the problem.

$\triangle$PH3C has a problem of using too much memory when enumerating the equality constraints. We ran 1,135 benchmark instances with 8GB of memory, and found that 473 instances ran out of memory to run *only pre-processing* on the CSP. We propose to change the representation of these generated equality constraints from a list of tuples to a bit matrix reduces the number of instances that ran out of memory to 387. We do so by following the same technique of extending $\triangle\text{PPC}^{bit}$ to $\triangle\text{PH3C}^{bit}$, by changing the representation of the equality constraints.

6.4.4 Decision Tree for Selecting Triangles for H3C

In Section 6.2.4 introduces a policy for choosing the triangles PPC should consider using the density of a primal graph. H3C operates on the dual graph, and such a policy ignores the differences between the two graphs. Figure 6.9 shows the decision tree we use for selecting triangles for H3C. Paraphrasing the content of Figure 6.9:

- We consider that, at a density of the primal graph of 50% or more (i.e., $d_p \geq 50\%$), AC fully propagates the impact of a variable instantiation. HLC typically yields only overhead but no further filtering. For this reason, we choose to simply enforce AC (see leaf 1 in Figure 6.9).
- In the remaining cases we choose to *always* exploit:

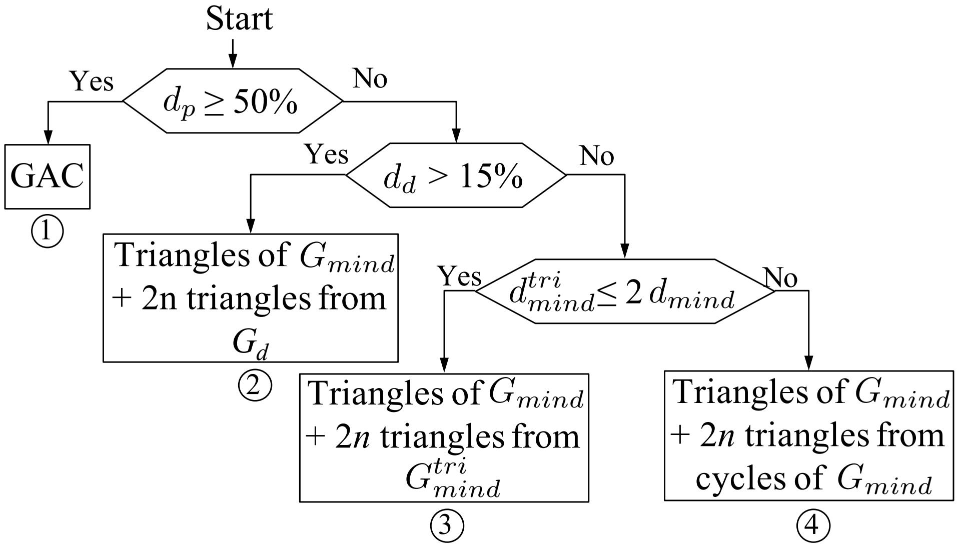


Figure 6.9: Selecting the triangles for PH3C

1. the existing triangles from a minimum dual graph G_{mind}. The triangles that appear in G_{mind} are analogous to the existing triangles for PPC as they are somewhat 'core' to the problem as redundant edges cannot transmit information.

2. $2n$ promising triangles that appear in the largest number of clusters in the tree decomposition (Section 6.2.2), generating the triangles on some version of the graph. For PH3C, the triangles are a combination of three relations, generated by triangulating the dual CSP. However, the tree decomposition is generated from the primal graph. To count the number of clusters a triangle appears in, we utilize each cluster not only contains a set of variables, but a set of relations.

- If the density of the dual graph is greater than 15% (i.e., $d_d \geq 15\%$), there is enough communication in the dual graph, and we choose to exploit $2n$ promising triangles from the dual graph (i.e., Triangles of G_d in leaf 2 in Figure 6.9).

- Otherwise we examine the triangulated minimal dual graph G^{tri}_{mind}.
- When G^{tri}_{mind} does not prohibitively add to the density of the minimal dual graph (i.e., $d^{tri}_{mind} \leq 2d_{mind}$), then we estimate that the triangulated minimal dual graph is not too dense and that the advantage of boosting propagation outweighs the overhead of increasing the number of triangles to process (i.e., Triangles of G^{tri}_{mind} in Figure 6.9)
- If G^{tri}_{mind} more than doubles the number of edges (i.e., $d^{tri}_{mind} > 2d_{mind}$), then the number of additional triangles resulting from triangulation can be overwhelming and may cause a serious overhead. We instead attempt to triangulate a minimal dual graph locally by first computing the minimum cycle basis, using our BFSC technique for finding approximation the union of cycles a graph node appears in (Section 5.3), and 'locally' triangulate the graph by triangulating the subproblem induced by the graph node and the returned cycles from BFSC. We select the subproblems from largest to smallest induced density. These triangles are called the triangles from the cycles of G_{mind} (i.e., Triangles from cycles of G_{mind} in Figure 6.9).

6.5 Empirical Evaluation of $\triangle$PH3C^{bit}

In this section, we evaluate the effectiveness of $\triangle$PH3C^{bit}. To this end, we consider the problem of finding a single solution to a CSP using backtrack search, the dom/wdeg variable ordering heuristic [Boussemart *et al.*, 2004], and real-full lookahead [Haralick and Elliott, 1980].

We first discuss our experimental setup. We then empirically study PH3C in three directions:

1. We compare the filtering strength of PH3C and PPC on binary CSPs.
2. We show the usefulness of the decision tree for selecting the triangles to use.
3. We compare the filtering strengths of PH3C.

Finally, we evaluate triggering $\triangle$PH3C using the when strategies PrePeak, BTWatch, and PP-BTWatch of Chapter 4. Importantly, we show the usefulness of enforcing $\triangle$PH3C during search.

6.5.1 Experimental Setup

We set up our experiments as follows. We use STR2+ [Lecoutre, 2011] as the GAC algorithm, which is always maintained during search. We use the $\triangle$PH3C^{bit} algorithm for enforcing PH3C, which we denote as $\triangle$PH3C^{bit} for simplicity.

We use the benchmark problems available from Lecoutre's website,[11] including all benchmarks with at least one instance with a primal graph of density less than 50%.[12] Indeed, on high density networks, the impact of an instantiation on a future variable is immediately propagated by GAC while HLC typically yields no further filtering but costs predictable data-setup overhead. This selection results in 137 benchmarks with 3,525 instances used in our experiments. The selected 137 benchmarks have a mixture of instances with densities $\geq 50\%$ and $< 50\%$, however, only 139 instances of the 3,525 instances included have densities $\geq 50\%$. We setup our decision tree (Section 6.2.4) to first compute the density of an instance. If the density is $\geq 50\%$, we enforce GAC. Otherwise, we execute the PH3C algorithm on the selected triangles. Our results include this computation time. We use a time limit of 60 minutes per instance and 8GB of memory.

---

[11] `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

[12] Table E.1 in Appendix E list the selected benchmarks.

In the tables that summarize our results, we report for each algorithm, where applicable:

- The number of instances solved in a given benchmark (#solved).
- The number of node visits averaged over the instances completed by all algorithms (avg. NV).
- The sum of the CPU time in seconds of the run time of an algorithm for all the instances in a benchmark completed by any of the compared algorithms ($\sum$CPU). When an algorithm does not terminate within the allocated time, we add 3,600 seconds to the CPU time and indicate with a '>' sign that the time reported is a lower bound.
- Finally, we highlight, with a boldface, the best value in a given row.

6.5.2 PH3C versus PPC on Binary CSPs

In Section 6.4.1, the definition of Hyper-3 Consistency (H3C) was extended to Conservative Hyper-3 Consistency (CH3C) and Partial Hyper-3 Consistency (PH3C) and were shown to be incomparable. We attempt to empirically quantify the strengths of PH3C and CH3C to PPC and CPC on binary CSPs by investigating the amount of tuples and values filtered at pre-processing. We report the number of instances that were found inconsistent at pre-processing (#Inconsistent). We report the difference in the filtering for each algorithm from the filtering of STR2+ (Tuples Rem. and Values Rem., respectively).

Table 6.7 compares the filtering of CH3C and PH3C on the dual graph, to that of CH3C and PH3C on a minimal dual graph, and to that of CPC and PPC. Although PH3C on the dual graph is the strongest, and thus filters the largest number of tuples

Table 6.7: Comparing the filtering obtained from PPC and PH3C

	Dual Graph		**Minimal Dual Graph**		**Primal Graph**	
	CH3C	**PH3C**	**CH3C**	**PH3C**	**CPC**	**PPC**
#Solved	1,995	525	2,445	1,305	2,487	2,388
$\sum$CPU	>1,897,317.2	>7,475,744.5	>226,548.9	>5,524,262.8	>120,314.2	>564,148.1
#Inconsistent	330	150	28	212	345	359
Avg. Tuples	2,579.0	10,336.2	2.0	8,127.6	1,065.8	14,236.1
Avg. Values	55.9	172.3	0.0	147.3	22.1	30.0
#Instances 2,776 total, 518 by all, 2,503 by at least one						

and values, it is able to complete the least number of instances (525). It is too powerful to be used in its full strength. On the other hand, reducing the property to filter on existing triangles of a minimum dual graph reduces the strength too much, filtering a surprisingly few number of tuples (2.0) and values (0.0).

Both PPC and CPC find more inconsistent instances than the H3C-type consistencies. This can partially be explained by the larger number of instances solved by CPC and PPC. Looking at the average number of values filtered, the dual-graph H3C techniques and PH3C on a minimal dual graph filter more values than PPC. However, PPC filters more tuples than these H3C techniques. Thus, the incompatibility between the techniques is prevalent and no one technique can be ruled better, in terms of strength.

6.5.3 Decision Tree for Selecting Triangles for PH3C

As shown in the previous section, there is a trade off between the CPU time and filtering power of PH3C and CH3C on the dual graph, as well as PH3C on a minimal dual graph. Thus, in this section, we evaluate the decision tree of Section 6.4.4. To that end, we enforce the resulting algorithm from each part of the decision tree at pre-processing, followed by search using GAC as RFL and the dom/deg ordering heuristic. We record the filtering on the search space by the reduction in the number of node visits. That is, we evaluate selecting $2n$ of the triangles from: the dual graph ($2n$ G_d), a triangulated minimal dual graph ($2n$ G^{tri}_{mind}), and the cycles of a minimal dual graph ($2n$ G^{cycle}_{mind}), and compare it against GAC and using the decision tree (DT). PH3C is enforced in all cases using the directional variant of PH3C (i.e., DPH3C). The choice of this variant is kept constant as a control, as the goal of this experiment is to assess the usefulness of the decision tree, not the various strengths.

Table 6.8 gives the overall performance on all the benchmarks evaluated. DPH3C using $2n$ G_d solved the largest number of instances (2,052), which is the only DPH3C technique to solve more than GAC (2,026). However, it is the DPH3C technique with the smallest reduction in node visits from GAC. DPH3C using $2n$ G^{tri}_{mind} had the largest reduction in node visits from GAC, but at the cost of the fewest number of instances solved.

Table 6.8: Enforcing the decision tree selections of PH3C at pre-processing followed by GAC as RFL

		DPH3C			
	GAC	**DT**	**$2n$ G^{cycle}_{mind}**	**$2n$ G_d**	**$2n$ G^{tri}_{mind}**
#Solved	2,026	1,946	1,917	**2,052**	1,665
∑CPU [sec]	>658,795.3	>1,078,761.1	>1,227,892.8	>**569,544.0**	>2,231,068.9
Avg. #NV	1,312,010.0	317,650.0	317,983.8	1,023,497.4	**283,113.7**
#Instances 3,241 total, 1,595 by all, 2,088 by at least one					

Of the instances that the decision tree solved (1,946), 64 were found inconsistent by GAC at pre-processing, thus, the decision tree did not make a decision. On the remaining instances, the decision tree selected $2n$ G^{cycle}_{mind} 1,573 times (83.6% of the time) $2n$ G_d 256 times (13.6%), and $2n$ G^{tri}_{mind} 53 times (2.8%). Thus, the decision tree emphasizes selecting the 'middle' strength (i.e., $2n$ G^{cycle}_{mind}), but occasionally selects the stronger (i.e., $2n$ G^{tri}_{mind}) or weaker (i.e., $2n$ G_d) strength. We evaluate using the decision tree in the remainder of this section, which we denote as $\triangle$PH3C$^+$.

6.5.4 Selecting PH3C Strength

We compare the various strengths of $\triangle$PH3C$^+$ (Section 6.4.2 by limiting the propagation queue and using the decision tree.

Table 6.9 shows the performance of running the $\triangle$PH3C$^+$ variants at pre-processing, while Table 6.10 shows the performance of running the $\triangle$PH3C$^+$ variants as RFL.

Table 6.9: $\triangle$PH3C$^+$ variants as pre-processing with dom/deg

Algorithm	**GAC**	**$\triangle$DH3C$^+$**	**$\triangle$DPH3C$^+$**	**$\triangle$P^{3}H3C$^+$**	**$\triangle$2DPH3C$^+$**	**$\triangle$PH3C$^+$**
#Solved	2,026	1,942	1,946	1,945	1,954	1,947
#Memout	734	756	756	756	756	756
$\sum$CPU [sec]	>633,595.3	>1,070,904.7	>1,053,561.1	>1,055,859.6	>1,032,102.7	>1,039,284.2
Avg. #NV	1,193,844.1	363,057.2	355,957.0	357,255.8	355,931.5	355,793.4
#Instances 3,780 total, 1,895 by all, 2,081 by at least one						

Table 6.10: $\triangle$PH3C$^+$ variants as RFL with dom/deg

Algorithm	**GAC**	**$\triangle$DH3C$^+$**	**$\triangle$DPH3C$^+$**	**$\triangle$P^{3}H3C$^+$**	**$\triangle$2DPH3C$^+$**	**$\triangle$PH3C$^+$**
#Solved	2,026	1,758	1,773	1,769	1,760	1,769
#Memout	734	756	756	757	758	756
$\sum$CPU [sec]	>608,395.3	>1,905,784.7	>1,887,383.1	>1,877,515.6	>1,898,658.9	>1,892,971.1
Avg. #NV	1,184,435.0	147,688.6	142,873.1	145,231.5	148,256.6	142,751.5
#Instances 3,780 total, 1,673 by all, 2,074 by at least one						

For both pre-processing and RFL, GAC performs better than any of the PH3C variations. This result is predictable given that PH3C is likely too strong for most problems, which we address in our triggering experiments in Section 6.5.5. The statistical rankings, according to a paired t-test, of the considered variations are as follows:

$$\text{pre-processing: } \triangle\text{2DPH3C}^+ \succ \triangle\text{PH3C}^+ \sim \triangle\text{DPH3C}^+ \succ \triangle\text{P}^3\text{H3C}^+ \sim \triangle\text{DH3C}^+$$

$$\text{RFL: } \triangle\text{P}^3\text{H3C}^+ \sim \triangle\text{DPH3C}^+ \sim \triangle\text{2DPH3C}^+ \sim \triangle\text{PH3C}^+ \succ \triangle\text{DH3C}^+$$

where $A \succ B$ denotes that A is statistically better than B and $A \sim B$ denotes that there are not statistically distinguishable.

6.5.5 $\triangle$PH3C$^+$ with PrePeak

We evaluate the use of PrePeak to trigger $\triangle$DH3C$^+$ using the decision tree (Section 6.4.4), and using the variation of DPH3C to iterate through the triangles only once (i.e., as a how much strategy). Table 6.11 compares the performance of DPH3C with GAC. $\triangle$DPH3C solves more instances than GAC in faster CPU time. Although it does have a few more memouts, the approach of using the decision tree and watching for memouts helps avoid them.

Table 6.11: Comparing GAC and PrePeak with $\triangle$DPH3C$^+$

	GAC	**$\triangle$DPH3C$^+$**
#Solved	1,254	**1,252**
#MemOut	58	85
$\sum$CPU [sec]	>215,914.2	**>213,326.7**
Avg. #NV	823,350.7	**225,780.4**
#Instances 2,126 total, 1,241 by all, 1,265 by at least one		

Table 6.12 highlights two exemplary benchmark for $\triangle$DPH3C: dubois and mug. The dual graph of the dubois benchmark is a ladder graph, thus by Theorem 16 it can be solved backtrack free if partial hyper-3 consistency is enforced. *This observation provides a graphical justification for why the benchmark is tractable.* Dubois is already

Table 6.12: Benchmarks where PrePeak with $\triangle$DPH3C$^+$ performs well

	GAC	**$\triangle$DPH3C$^+$**
dubois #Instances 13 total, 6 by all, 13 by at least one		
#Solved	6	**13**
#MemOut	0	0
$\sum$CPU [sec]	>29,484.0	**0.5**
Avg. #NV	123,405,942.7	**0**
mug #Instances 8 total, 4 by all, 8 by at least one		
#Solved	6	**13**
#MemOut	0	0
$\sum$CPU [sec]	>14,400.1	**721.3**
Avg. #NV	94.0	**94.0**

known to be a tractable benchmark by looking at its constraint semantics, meaning the definition of the constraint, as because all of the constraints can be re-written as implication constraints (i.e., ⇔) [Ostrowski *et al.*, 2002]. As for the mug benchmark, DPH3C is not able to solve the instance backtrack free, but it must search. However, the high strength of △DPH3C was able to shrink the search space to allow all of the instances of mug to be solved.

As for the other benchmarks, △DPH3C performs similarly to GAC, provided it does not memout. Indeed, PrePeak triggers △DPH3C few times as it filters relatively few domain-values given the amount of effort it takes to enforce it. This result can be explained by that △DPH3C not only needs to filter both the equality constraints, and the CSP constraints, before it is able to filter any domain values. The amount of 'indirection' between the equality constraints and the CSP variables hinders its ability to filter many values.

Summary

In this chapter we studied a special form of cycles: triangles. Partial Path Consistency (PPC) takes advantage of triangles, especially the △PPC algorithm, thus we empirically evaluate △PPC as lookahead, which has never been studied. Further, we presented the first algorithm for enforcing Partial Hyper-3 Consistency (PH3P) by adapting △PPC to △PH3P and empirically evaluate it as lookahead. We notice that the dubois benchmark can be determined inconsistent at pre-processing by PH3P, and identify the structural property (Theorem 16 of Chapter 5) to explain its tractability.

Chapter 7

Conclusions and Future Work

This chapter concludes the dissertation by summarizing our contributions and giving directions for future research.

7.1 Summary of Contributions

Constraint Satisfaction Problems (CSPs) are usually solved with search. To reduce the size of the search space, backtrack search is typically interleaved with constraint propagation. Stronger consistency algorithms can filter larger portions of the search space at the cost of an increased CPU time.

The research presented in this dissertation addresses the question of enforcing high-level consistency during search. We offer a new perspective that characterizes the various possible approaches in term of when, where, and how much of a higher-level consistency to enforce during search.

Figure 7.1 gives an overview of the different when, where, and how much strategies advocated for in this thesis. In particular, Chapter 4 introduces PrePeak as a strategy for determining *where* to enforce higher-level consistency (HLC) depending

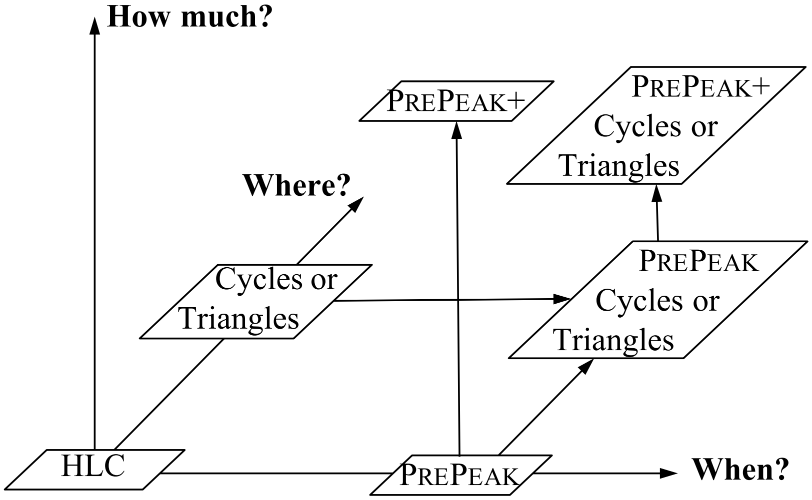


Figure 7.1: The dimensions of enforcing consistency investigated in this dissertation

on the number of backtracks. PrePeak$^+$ combines this 'where strategy' with a *how much* strategy to interrupt the consistency algorithm after processing a given number of elements in the algorithm's propagation queue or after a given CPU time has passed. Chapter 5 and Chapter 6 localize the operations of the consistency algorithms to cycle structures of the CSP and to triangles, respectively. They also combine the resulting new consistencies with PrePeak and PrePeak$^+$.

In summary, this dissertation introduces a framework for enforcing higher-level consistency on a CSP that adapts its filtering to the problem at hand.

7.2 Directions for Future Research

Below we identify directions for further research, which are beyond the scope of this dissertation:

1. *Adjusting triangles using the clusters of a tree decomposition to include all variables*: The strategy for selecting triangles depending on the number of clusters of a tree decomposition they appear in (Section 6.2.2) may allow for a variable

to not participate in any selected triangles. We propose to add an additional step after selecting the triangles to check whether or not a variable appears in at least one selected triangle. If it does not, but the variable appears in some non-generated triangle, we will accept that triangle too. This process will ensure that every variable appears in some selected triangle.

2. *Dynamically adjusting powers of r in* PREPEAK: In Section 4.1.2 we advocate for using $(r_w, r_f, r_n) = (r^{-1}, r^2, r^3)$. However, we may want to adjust the values of these powers depending on the progress of search. For example, we may have the following 'policies:'

   - $A = (1/r, 1, r)$,
   - $B = (1/r, r, r^2)$,
   - $C = (1/r, r^2, r^3)$

   We start search using policy A, which is the most aggressive and will apply HLC a lot. We can detect the deepest depth that search reaches, and change to policy B at some point, possibly looking at the total cumulative time of HLC and HLC switching once HLC takes more time than GAC. Repeat running with policy B, comparing the maximum depth reached in this policy to determine if we should return to policy A, or go down to policy C, which is the more conservative.

3. *Applying trigger for other consistencies*: We have validated the trigger strategy in the context of GAC versus POAC (which is a variable-based consistency), and GAC versus PPC and GAC versus PH3C (which are relational consistencies). We can extend our approach to other high-level consistencies, in particular, RNIC [Woodward *et al.*, 2011b].

4. *Extending cycles to (Relational) Neighborhood Inverse Consistency*: Woodward *et al.* [2011b] proposed four strengths of Relational Neighborhood Inverse Consistency (RNIC), where the neighborhood of a variable is adjusted depending on the dual graph used (i.e., the dual graph, a minimal dual graph, the triangulated dual graph, or a triangulated minimal dual graph). The neighborhood could be determined by using a minimal cycle basis. Such a selection of the neighborhood may also be useful for Neighborhood Inverse Consistency [Freuder and Elfe, 1996].

5. *Applications of counting backtracks*: Epstein *et al.* [2002] studied the use of different variable and value-ordering heuristics at different depths of the search tree. They identify three static categorizations of the location of search, depending on the amount of variables assigned:

   a) early in search tree, fewer than 20% of the variables assigned,
   b) middle of the search tree, at least 20% but no more than 80% of the variables assigned, and
   c) late in the search tree, more than 80% of the variables assigned.

   Instead of relying of static percentages to determine the levels for switching heuristics, we propose to use the count the number of backtracks per depth (BpD).

6. *Improving variable ordering using ghost constraints*: In the experimental analysis of PPC (Section 6.3), we do not allow the added constraints to adjust the degree of a variable in dom/(w)deg. We propose to study using the added constraints by MinFill, which we call adding 'ghost' constraints, in a new degree-based ordering heuristic. Indeed, we found out that, for some benchmarks,

ghost constraints are extremely useful for improving the ordering heuristic. For example, jobShop is one particular benchmark where ghost constraints are effective. Allowing AC to operate on the added constraints, filtered by PPC, has benefits, but not as dramatic as when used with the ordering heuristic.

Another way of looking at ghost constraints in the context of dom/wdeg is that the ghost constraints provide a sort of initialization of the 'wdeg.' Because the ghost constraints are not apart of the problem, they cannot cause a wipeout, and thus, cannot have their 'ghost' weight updated.

7. *Combining hyper-3 consistency with pair-wise consistency*: Hyper-3 Consistency (H3C) modifies the dual constraints in the dual CSP. Pair-wise consistency (PWC) exploits the dual constraints in filtering the constraints of the CSP. Because PWC is cheaper to enforce than H3C, it may be advantageous to exploit running PWC prior to enforcing H3C. However, current PWC algorithms exploit the equality property of the dual constraints and H3C algorithms 'break' this equality. Either a new PWC algorithm should be created to exploit the filtering of H3C, or the PWC would run ignoring the filtering of H3C, possibly reducing the effectiveness of the PWC algorithm.

8. *Conservative dual consistency-like algorithm for hyper-3 consistency*: Develop a Conservative Dual Consistency (CDC)-like property and algorithm for enforcing a Hyper-3 Consistency (H3C)-like consistency on a CSP (i.e., run a CDC algorithm on the dual CSP). On a binary CSP, CDC looks at the two vvp's $(V_i, a), (V_j, b)$ and requires

$$(C_{i,j} \notin C) \vee ((V_j, b) \in AC(P|V_i = a) \wedge (V_i, a) \in AC(P|V_j = b))$$

$AC()$ would mean $PWC()$ for the dual CSP. But, then we *need* a way to enforce PWC using the 'filtered' equality constraints, because it operates only by enforcing PWC (otherwise the filtered constraints are never used for anything).

Note that such an algorithm cannot use a PWC algorithm that exploits the piecewise functionality of the equality constraints of the dual CSP. Maybe one could use the CT GAC-algorithm [Demeulenaere *et al.*, 2016] on the dual graph to that end.

9. *New heuristics for prioritizing singleton tests*: Adaptive POAC is a strategy for early termination of the POAC-1 algorithm depending on the effectiveness of the singleton tests [Balafrej *et al.*, 2014]. The variables to singleton test are prioritize using the variable ordering heuristic dom/wdeg. The heuristics of Stergiou [2008] for switching between GAC and maxRPC for binary CSPs and Paparrizou and Stergiou [2012] between GAC and maxRPWC for nonbinary CSPs could be used for prioritizing variables. In particular, the variables on which they enforce maxRPC/maxRPWC could be the variables used for doing the singleton test. The use of different heuristics for ordering the singleton tests for POAC also impacts the use of PrePeak$^+$ with POAC, as it only visits part of the propagation queue.

In conclusion, this dissertation has positively answered our original question to provide a strategy to determine where, when, and how much of a higher-level consistency to enforce during search. Further, it has opened up new directions for further research.

Appendix A

Weight-Based Variable Ordering in the Context of High-Level Consistency

Dom/wdeg is one of the most effective heuristics for dynamic variable ordering in backtrack search [Boussemart *et al.*, 2004]. As originally defined, this heuristic increments the weight of the constraint that causes a domain wipeout (i.e., a dead-end) when enforcing arc consistency during search. "The process of weighting constraints with dom/wdeg is not defined when more than one constraint lead to a domain wipeout [Vion *et al.*, 2011]." In this chapter, we investigate how weights should be updated in the context of two high-level consistencies, namely, singleton (POAC) and relational consistencies (RNIC). We propose, analyze, and empirically evaluate several strategies for updating the weights. We statistically compare the proposed strategies and conclude with our recommendations.

A.1 Motivation

Variable-ordering heuristics are critical for the effectiveness of backtrack search to solve Constraint Satisfaction Problems (CSPs). Common heuristics implement the fail-first principal, choosing the most constrained variable as the next variable to assign. One such heuristic is dom/ddeg, which selects the variable with the smallest ratio of its current domain to its future degree. A more recent heuristic, dom/wdeg, uses the weighted degree of a variable by assigning a weight, initially set to one, to each constraint, and incrementing this weight whenever the constraint causes a domain wipeout [Boussemart *et al.*, 2004]. Recently, higher-level consistencies (HLC) have shown promise as lookahead for solving difficult CSPs [Bennaceur and Affane, 2001; Woodward *et al.*, 2011b; Woodward *et al.*, 2012; Balafrej *et al.*, 2014].

Because HLC algorithms typically consider more than one constraint at the same time, updating the weights of the constraints in dom/wdeg is currently an open question [Vion *et al.*, 2011]. This chapter focuses on answering this question in the context of two high-level consistencies, namely, Partition-One Arc-Consistency (POAC) [Bennaceur and Affane, 2001] and Relational Neighborhood Inverse Consistency (RNIC) [Woodward *et al.*, 2011b]. Our study focuses on these two consistencies because they have both been shown to be beneficial when used for lookahead during search.

For POAC and RNIC we introduce four and three strategies, respectively, to increment the weights of the constraints. For both consistencies we find that a baseline strategy corresponding to the original dom/wdeg proposal is statistically the worst of the proposed strategies. We conclude the high-level consistency should influence the weights. For POAC we find that the proposed strategy AllS is statistically the best. For RNIC the two non-baseline strategies are statistically equivalent.

Other popular variable-ordering heuristics include Impact-Based Search [Refalo, 2004] and Activity-Based Search [Michel and Van Hentenryck, 2012]. These heuristics rely on information about the domain filtering resulting from enforcing a given consistency. Because they ignore the operations of the consistency algorithm, it is not clear how these heuristics could be used to order the propagation queue of the consistency algorithm [Wallace and Freuder, 1992; Balafrej *et al.*, 2014]. Further, it is also not clear how to apply them in the context of consistency algorithms that filter the relations [Woodward *et al.*, 2011b; Woodward *et al.*, 2012].

In this chapter, we introduce our weighting schemes for POAC and RNIC and then empirically evaluate them.

A.2 Weighting Schemes

We introduce weighting schemes first in the context of singleton consistencies, namely Partition-One Arc-Consistency (POAC), and then in that of relational consistencies, namely Relational Neighborhood Inverse Consistency (RNIC).

Enforcing a high-level consistency (HLC) property is typically costlier than enforcing GAC, but typically yields more powerful pruning. Further, it is often more effective, in terms of CPU time, to run a GAC before an HLC algorithm [Debruyne and Bessière, 1997b], as we choose to do in this chapter.

A.2.1 Partition-One Arc-Consistency (POAC)

We first investigate the case of POAC, which operates by initially running a GAC algorithm then applying the following operation to each variable until no change occurs. For a given variable, it applies a singleton test to each value in the domain of the variable. A singleton test assigns the value to the variable and enforces GAC on the problem. We propose four strategies to increment weights during POAC:

OLD: We allow only the GAC call before POAC to increment the weight of the constraint that causes a domain wipeout. That is, POAC is not allowed to alter the weights. This strategy is the simplest and it is a direct application of the original proposal [Boussemart *et al.*, 2004]. In our experiments we use this strategy as a baseline and show it does not perform well in practice.

ALLS: In addition to incrementing the weights according the above strategy (i.e., OLD), we allow every singleton test to increment the weight of a constraint whenever enforcing GAC on this constraint during the singleton test directly wipes out the domain of a variable. This update is made at most once for each singleton test. Under this strategy, all constraints that caused domain wipeouts are affected, thus, we call it ALLS. Notice that the weight of more than one constraint may be updated even though search does not have to backtrack. This behavior differs from the original proposal [Boussemart *et al.*, 2004].

LASTS: In addition to incrementing the weights according to OLD, we increment the weight of the constraint causing a domain wipeout at the *last* singleton test on a given variable if and only if all previous singleton tests on the values of this variable have failed. Thus, we only increment the weight of a single constraint and do so only when search has to backtrack, which conforms to the spirit of the original heuristic. Notice, the order of values singleton tested affects this strategy.

VAR: This strategy encapsulates OLD as a first step and increments the weight of the *variable* on which all singleton tests have failed (thus forcing search to backtrack). In order to implement this strategy we add a counter for the weight of each variable w_v, initially zero. When a variable fails all of its singleton tests during propagation the counter w_v for that variable is incremented by one.

We propose to integrate w_v with the weighted degree function of dom/wdeg as follows:

$$\alpha_{wdeg}^{\textsc{Var}}(x_i) = w_v(x_i) + \sum_{(c \in \mathcal{C}_f) \wedge (x_i \in scp(c))} w_c(c) \tag{A.1}$$

where $\mathcal{C}_f \subseteq \mathcal{C}$ is the set of constraints with at least two future variables. The rationale behind this strategy is the following. The goal of the heuristic dom/wdeg is to identify the conflicts in the problem and address them earlier, rather than later, in the search. VAR puts the blame on the variable that first caused the failure of POAC.

A.2.2 Relational Neighborhood Inverse Consistency (RNIC)

The relational consistency property RNIC is equivalent to enforcing Neighborhood Inverse Consistency (NIC) on the dual graph of the CSP [Freuder and Elfe, 1996; Woodward *et al.*, 2011b]. The RNIC property ensures that every tuple in every relation can be extended to a solution in the subproblem induced on the dual graph of the CSP by the relation and its neighboring relations. The RNIC algorithm operates on table constraints and removes, from a given relation, all the tuples that do not appear in a solution in the induced (dual) CSP of its neighborhood [Woodward *et al.*, 2011b]. We propose three strategies to increment weights when RNIC is used for lookahead during search:

OLD: As in POAC in Section A.2.1, we allow only the GAC call (preceding the call to RNIC) to increment the weight of the constraint that causes domain wipeout.

ALLC: This strategy encapsulates OLD as a first step. During lookahead, RNIC is called on each constraint with two or more future variables. When the RNIC algorithm removes all the tuples of a given relation, AllC increments the weights of all the relations in the induced (dual) CSP. The rationale being that this considered combination of relations (which is the relation and its neighborhood in the dual graph) is ‘collectively’ responsible for the ‘relation’ wipeout.

Head: This strategy is similar to AllC, except that we increment only the weight of the constraint whose relation was emptied by the RNIC algorithm and do not increment the weights of its neighborhood in the dual graph.

A.3 Experimental Evaluation

We evaluate the effectiveness of the strategies proposed for POAC and RNIC in Sections A.3.2 and A.3.3, respectively.

A.3.1 Experimental Setup

We consider the problem of finding a single solution to a CSP using backtrack search with some lookahead, d-way branching, dom/wdeg dynamic variable-ordering heuristic [Boussemart *et al.*, 2004], and lexicographic value ordering. We use STR2+ for enforcing GAC [Lecoutre, 2011], APOAC for enforcing POAC [Balafrej *et al.*, 2014],[1] and selRNIC for enforcing RNIC [Woodward *et al.*, 2011b]. We use the benchmark problems available from Lecoutre's website.[2] Benchmarks are selected separately for POAC and RNIC. For a given consistency level, if any instance is solved by any of the weighing schemas of the considered consistency within the time limit of 60 minutes

---

[1] Using the terminology of Balafrej et al. [Balafrej *et al.*, 2014], we use the following parameters and their recommended values for APOAC $maxK = n$, last drop with $\beta = 0.05$, and 70%-PER. Where $maxK$ indicates the number of processed items in the propagation queue, β is the threshold of search-space reduction during the learning phase and 70%-PER is the percentile for learning the value of $maxK$.

[2] `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

and memory limit of 8GB, then the entire benchmark is included in the experiment. For benchmarks in intension we convert the instance to extension prior to solving and do not include the time for conversion.[3] From the 254 benchmark problems (total 8,549 instances) available on Lecoutre's website, our results are reported on 144 benchmarks (total 4,233 instances) for POAC and 132 (total 3,869 instances) for RNIC.

We summarize the results of these experiments in Tables A.2–A.7 and Figures A.1 and A.2. For each strategy, we report in Tables A.2–A.7:

- The number of completions (# Completions) with the total number of instances in parenthesis.
- The sum of the CPU time in seconds ($\sum$CPU sec.) computed over instances where at least one algorithm terminated (given in parenthesis). When an algorithm does not terminate within 60 minutes, we add 3,600 seconds to the CPU time and indicate with a > sign that the time reported is a lower bound. We boldface the smallest CPU time.
- The average number of node visits (Average NV) computed over the instances where all strategies completed (given in parenthesis).

Figures A.1 and A.2 plot the number of instances solved by each strategy (Y-axis) as the CPU time increases (X-axis).

In addition to the above experiment, we also conduct a statistical analysis of the relative performance of the proposed strategies. We compare pairwise the strategies corresponding to each higher-level consistency (i.e., POAC and RNIC) in order to

---

[3] In a study not reported we found that STR2+ is faster at solving CSP instances than running GAC on the original intension constraints because STR explores the satisfying tuples instead of valid tuples. As STR and RNIC algorithms require table constraints we pre-convert the instances. The conversion time is the same for each algorithm and can safely be ignored.

determine whether or not a statistical difference exists between the strategies. Because search may fail to complete within the time limit, we consider our results to be right-censored and analyze them using a nonparameterized Wilcoxon signed-rank test [Wilcoxon, 1945]. The test operates by comparing the rank of the differences of the paired data. Differences of zero have no effect on the test and are safely discarded before ranking. Further, given the clock precision, we discard data points where the CPU difference is less than one second. We assume a one-tailed distribution and significance level of $p = 0.05$.[4] In the presence of censored data, we adopt the following procedure to generate the data for each pairwise test. First, we run each strategy on each instance for the time limit (i.e., 60 minutes). If both strategies solve the instance, the data is included in the analysis. If neither strategy solves the instance, the instance is excluded from the analysis (i.e., the difference is zero and discarded). If one strategy completes within the time threshold and the other does not, we re-run the second strategy with double the time limit (i.e., 120 minutes), recording this limit as the completion time in case search does not terminate earlier. By allowing the additional time, the censored data no longer affects the significance of the analysis [Palmieri *et al.*, 2016].[5] The results obtained with the doubled time limit are used only for the statistical analysis ranking the relative performance of the strategies (Table A.1 and Expression (A.2)), but not used for the results reported in Tables A.2–A.7.

---

[4] Check Palmieri et al. [Palmieri *et al.*, 2016] for an overview of the Wilcoxon signed-rank test and the adopted methodology.

[5] Our approach is similar to that of Palmieri et al. [Palmieri *et al.*, 2016] except that we exclude instances that neither strategy completes with the original time limit.

Table A.1: Statistical analysis of weighting schemes for POAC

Benchmark	**Ranking**						
All benchmarks, put together	AllS	>	LastS	≡	Var	>	Old
‘QCP/QWH,’ ‘BQWH’ (quasi-group completion)	LastS	>	Old	>	AllS	≡	Var
‘Graph Coloring’	Var	>	AllS	>	LastS	>	Old
‘RAND’ (random)	Var	>	AllS	≡	LastS	≡	Old
‘Crossword’	Var	>	AllS	≡	LastS	≡	Old

A.3.2 Partition-One Arc-Consistency

Based on the statistical analysis comparing the relative performance for Old, AllS, LastS, and Var for POAC, we conclude that *overall* (Table A.1):

- AllS outperforms all others strategies
- LastS and Var are equivalent
- Old exhibits the worst performance of the four strategies, showing that it is important for dom/wdeg to increment the weights with POAC, which justifies our investigations.

However, a careful study of the individual benchmarks shows that LastS on many quasi-group completion benchmarks and Var are competitive on many, but *not* all, graph coloring, random, and crossword benchmarks.[6] Re-running the statistical analysis on each group of those benchmarks yields the results shown in the last four rows of Table A.1. Again, we insist that even when considering individual benchmarks, the performance of AllS remains *globally* the most robust and consistent of all four strategies.

---

[6] Using the categories identified on Lecoutre’s website.

Table A.2 summarizes the experiments' results on the 144 tested benchmarks. In terms of the number of completed instances and the CPU time, AllS is the best (with 2,822 instances and >1,033,699 seconds) and Old is the worst (with 2,804 instances and >1,139,552 seconds) of the four proposed strategies. In terms of the average number of nodes visited (i.e., reduction of the search space), LastS visits the least amount of nodes on average (16,503), followed by AllS (16,712), Old (19,181), and Var (21,875).[7]

Table A.2: Overall results of experiments for POAC

	Old	AllS	LastS	Var
Completion (4,233)	2,804	2,822	2,814	2,811
$\sum$CPU sec. (2,846)	>1,139,552	**>1,033,699**	>1,075,640	>1,065,547
Average NV (2,775)	19,181	16,712	16,503	21,875

Table A.3 summarizes individual benchmark results for the quasi-group completion category. Compared to the quasi-group completion analysis in Table A.1, the benchmarks typically follow the statistical trend with LastS performing the best on the QCP-15 and QWH-20 benchmarks. However, although LastS was statistically the best, on bqwh-15-106, AllS was the fastest.

Table A.4 summarizes individual benchmarks for graph coloring, random, and crossword benchmarks. For these categories of benchmarks the statistical analysis of Table A.1 shows that Var performs the best. Indeed, for full-insertion, tightness0.8, and wordsVg Var has the smallest CPU time of the strategies. However, individual

---

[7] We offer the following hypothesis as to why Var has the largest average of nodes visited. The heuristic dom/wdeg is a 'conflict-directed' heuristic in that it attempts to select the variable that participates in the largest number of 'wipeouts.' By incrementing the weight of the variable being singleton-tested, Var perhaps increases the importance of a variable that 'sees' the conflict rather than those variables that 'cause' the conflict. This hypothesis deserves a more thorough investigation.

Table A.3: Examples of quasi-group completion benchmark for POAC

Benchmark		Old	AllS	LastS	Var
	Where LastS *performs best*				
	Completion (15)	15	15	15	15
QCP-15	∑CPU sec. (15)	3,920	5,480	**3,214**	6,083
	Average NV (15)	30,488	38,641	23,963	33,589
	Completion (10)	9	9	9	9
QWH-20	∑CPU sec. (9)	6,625	7,329	**5,631**	12,337
	Average NV (9)	57,453	58,623	45,095	63,225
	. . . but AllS *can still win on such benchmarks*				
	Completion (100)	100	100	100	100
bqwh-15-106	∑CPU sec. (100)	196	**167**	189	211
	Average NV (100)	599	433	531	507

Table A.4: Examples of graph coloring, random, crossword benchmarks for POAC

Benchmark		Old	AllS	LastS	Var
	Where Var *performs best*				
	Completion (41)	28	28	28	29
full-insertion	∑CPU sec. (29)	>12,720	>10,055	>10,182	**7,238**
	Average NV (28)	16,725	12,676	13,312	8,749
	Completion (100)	98	97	97	99
tightness0.8	∑CPU sec. (99)	>59,907	>53,042	>56,945	**41,848**
	Average NV (97)	1,213	1,085	1,196	1,315
	Completion (65)	55	56	54	59
wordsVg	∑CPU sec. (59)	>24,376	>24,190	>28,533	**17,913**
	Average NV (54)	298	391	411	250
	. . . but AllS *can still win on such benchmarks*				
	Completion (26)	20	20	20	20
sgb-book	∑CPU sec. (20)	9,677	**8,315**	8,455	8,565
	Average NV (20)	143,653	148,055	148,985	134,099
	Completion (100)	100	100	100	100
tightness0.1	∑CPU sec. (100)	46,926	**43,766**	44,971	69,974
	Average NV (100)	10,347	9,762	9,948	12,457
	Completion (65)	29	31	28	30
ukVg	∑CPU sec. (31)	>19,466	**19,040**	>20,961	>19,119
	Average NV (28)	141	411	133	139

benchmarks may vary despite the identified statistical groupings. For example, AllS performs best on the tightness0.1, sgb-book, and ukVg benchmark, respectively.

We conclude that, unless we know enough about the problem instance under consideration, we should use AllS in conjunction with POAC, as the overall analysis shows us.

Figure A.1 shows the cumulative number of instances completed by each strategy as CPU time increases. For easy instances (< 100 seconds), the completions of the strategies are similar. As the time limit increases Old becomes dominated by the other three strategies. To better compare AllS, LastS, and Var we examine the hard instances, zooming the chart on the cumulative CPU time solved between 1,000 and 3,600 seconds. Although Var performs well on smaller CPU time (Var contends with AllS for the most completed instances between 1,000 and 1,700 seconds) it

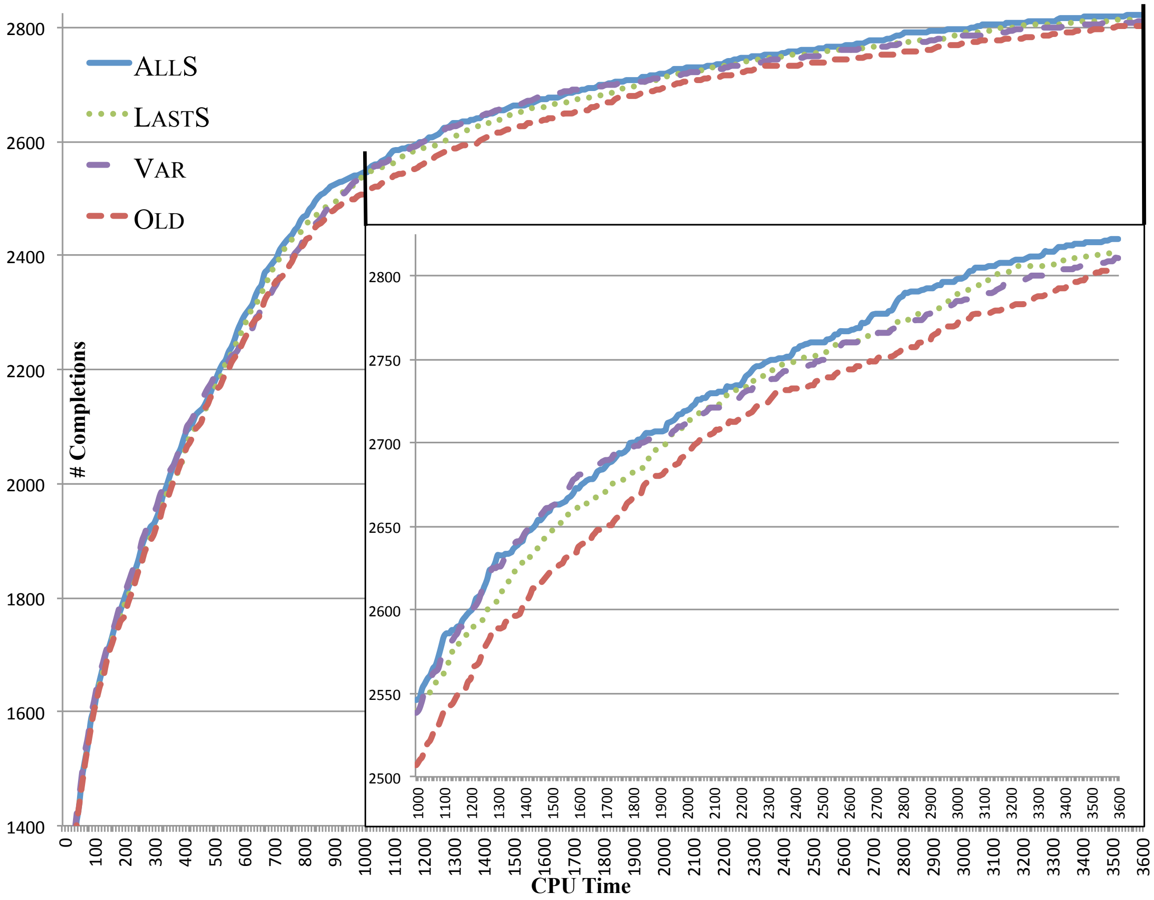


Figure A.1: Cumulative number of instances completed by CPU time for POAC becomes dominated by AllS and LastS on the harder instances. AllS clearly dominates all other strategies. These curves confirm the results of the statistical analysis given in Table A.1.

A.3.3 Relational Neighborhood Inverse Consistency

The statistical analysis compares the relative performance for Old, AllC, and Head for RNIC. It shows that, *overall*, AllC and Head are equivalent and Old has the worst performance. The following holds in general for all benchmarks:

$$\text{AllC} \equiv \text{Head} > \text{Old} \tag{A.2}$$

The fact that Old is the worst demonstrates that RNIC's contribution to the weights of dom/wdeg should not be ignored, thus justifying our investigations.

Table A.5 summarizes the experiments' results on all the 132 tested benchmarks. AllC is the best strategy on all measures while Old is the worst.

Table A.5: Results of experiments for RNIC

	Old	AllC	Head
# Completion (3,869)	2,420	2,427	2,423
$\sum$CPU sec. (2,416)	>1,032,130	>**1,010,221**	>1,014,635
Average NV (2,432)	77,067	45,696	45,803

We were not able to uncover meaningful categories of benchmarks to distinguish between AllC and Head. Table A.6 summarizes individual benchmark results for the Dimacs category. Within the category, either AllC or Head perform the best by all measures on different benchmarks. Similar results are obtained on the graph coloring category, shown in Table A.7. Having such different results between AllC and HEAD explains why the statistical analysis found them to be equivalent. Regardless, either ALLC or HEAD performs better than OLD in a statistically significant manner.

Figure A.2 shows the cumulative number of instances completed by each strategy as CPU time increases. As was the case for POAC, on easy instances (< 100 seconds),

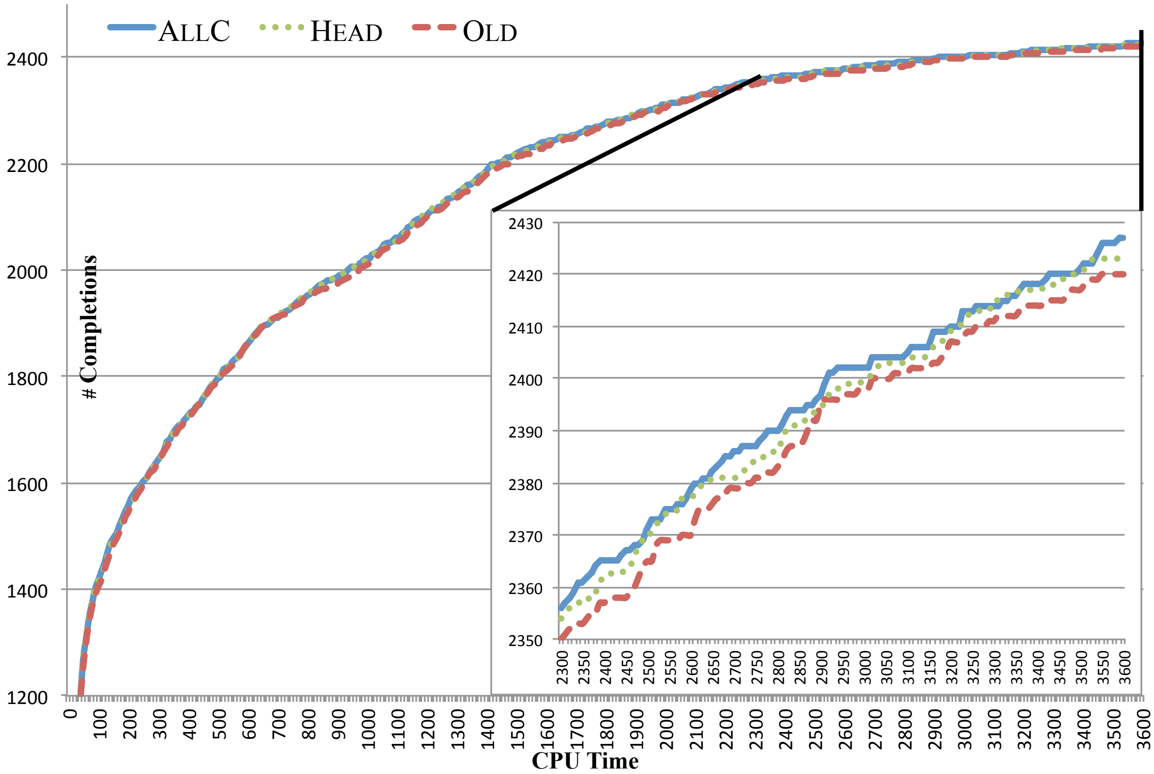


Figure A.2: Cumulative number of instances completed by CPU time for RNIC

Table A.6: Examples of Dimacs benchmarks where ALLC and HEAD perform best

Benchmark		OLD	ALLC	HEAD
pret	Completion (8)	4	4	4
	ΣCPU (4)	196	**28**	61
	Average NV (4)	1,285,234	125,793	273,736
dubois	Completion (13)	6	9	11
	ΣCPU (6)	>22,041	>10,088	**1,348**
	Average NV (11)	11,222,349	1,522,902	382,329

Table A.7: Two graph coloring benchmarks where AllC and Head perform best

Benchmark		Old	AllC	Head
mug	Completion (8)	8	8	8
	ΣCPU (8)	5,098	**548**	2,819
	Average NV (8)	1,501,379	189,595	883,130
leighton-15	Completion (26)	5	5	5
	ΣCPU (5)	2,219	1,493	**1,222**
	Average NV (5)	25,014	12,461	4,972

the completions of the strategies are similar. Focusing on harder instances, solved between 2,300 and 3,600 seconds, Old becomes dominated by AllC and Head. The curves of AllC and Head remain close to one another. These curves confirm the ranking in Equation A.2.

Summary

This chapter introduces four strategies for incrementing the weight in dom/wdeg for singleton consistencies (POAC) and three strategies for relational consistencies (RNIC). For both consistencies, Old is the worst strategy and a weighting schema involving the higher-level consistency is necessary. We show that for POAC the best method is AllS, which increments the weights at every singleton test. For RNIC, we show AllC and Head are statistically equivalent. Our work is a first step in the right direction, especially given the importance of higher-level consistencies in solving difficult CSPs. Future work may need to investigate more complex strategies for these and other consistencies.

Appendix B

Adaptive Parameterized Consistency for Non-Binary CSPs by Counting Supports

Determining the appropriate level of local consistency to enforce on a given instance of a Constraint Satisfaction Problem (CSP) is not an easy task. However, selecting the right level may determine our ability to solve the problem. Adaptive parameterized consistency was recently proposed for binary CSPs as a strategy to dynamically select one of two local consistencies (i.e., AC and maxRPC). In this chapter, we propose a similar strategy for non-binary table constraints to select between enforcing GAC and pairwise consistency. While the former strategy approximates the supports by their rank and requires that the variables domains be ordered, our technique removes those limitations. We empirically evaluate our approach on benchmark problems to establish its advantages. This work has been published [Woodward *et al.*, 2014].

B.1 Introduction

There is an abundance of local consistency techniques of varying cost and pruning power to apply to a Constraint Satisfaction Problem (CSP), but choosing the right one for a given instance remains an open question. In a portfolio approach [Xu *et al.*, 2008; Kadioglu *et al.*, 2011; Geschwender *et al.*, 2013], we typically choose a single consistency level and enforce it on the entire problem (or a subproblem). Heuristic-based methods have been proposed to dynamically switch, at various stages of search and depending on the constraint, between a weak and a strong level of consistency, AC and maxRPC for binary CSPs [Stergiou, 2008] and GAC and maxRPWC for non-binary CSPs [Paparrizou and Stergiou, 2012]. The above-mentioned approaches do not allow us to enforce different levels of consistency on the values in the domain of the same variable. To this end, Balafrej et al. introduced *adaptive parameterized consistency*, which selects, for each value in the domain of a variable, one of two consistency levels based on the value of a parameter [Balafrej *et al.*, 2013]. That parameter is determined by the *rank* of the support of the value in a constraint (assuming a fixed total ordering of the variables' domains), and updated depending on the weight of the constraint [Boussemart *et al.*, 2004]. Their study targeted enforcing AC and maxRPC on binary CSPs.

In this chapter, we extend their mechanism to enforcing GAC and pairwise-consistency on non-binary CSPs with table constraints. Our approach is based on *counting* the number of supporting tuples, which is automatically provided by the algorithms that we use. Thus, we remove the restriction on maintaining ordered domains and the approximation of a support's count by its rank. We establish empirically the advantages of our approach.

B.1.1 Local Consistency Properties

CSPs are typically solved with backtrack search. To reduce the severity of the combinatorial explosion, CSPs are usually filtered by enforcing a given *local consistency property* [Bessiere, 2006].

A variable-value pair $\langle x_i, v_i \rangle$ has an arc-consistent support (AC-support) $\langle x_j, v_j \rangle$ if the tuple $(v_i, v_j) \in R_{ij}$ where $\mathit{scope}(R_{ij}) = \{x_i, x_j\}$ [Mackworth, 1977; Bessière *et al.*, 2005]. A CSP is arc consistent if every variable-value pair has an AC-support in every constraint. Generalized Arc Consistency (GAC) generalizes arc consistency to non-binary CSPs [Mackworth, 1977]. $\langle x_i, v_i \rangle$ has a GAC-support in constraint c_j if $\exists \tau \in R_j$ such that $\tau[x_i] = v_i$. A CSP is GAC if every $\langle x_i, v_i \rangle$ has a GAC-support in every constraint in $\mathit{cons}(x_i)$. GAC can be enforced by removing domain values that have no GAC-support, leaving the relations unchanged. Simple Tabular Reduction (STR) algorithms not only enforce GAC on the domains, but also remove all tuples $\tau \in R_j$ where $\exists x_i \in \mathit{scope}(R_j)$ such that $\tau[x_i] \notin \mathit{dom}(x_i)$ [Ullmann, 2007; Lecoutre, 2011; Lecoutre *et al.*, 2012].

The STR and STR2(+) algorithms use two data-structures to maintain the alive set of tuples in a constraint c_i, $\mathit{currentLimits}[c_i]$, and $\mathit{position}[c_i]$. These data structures allow easy restoration of tuples upon backtrack during search [Lecoutre, 2011; Ullmann, 2007]. In STR3, unnecessary traversals of the relation is avoided by recording for each $\langle x_i, v_i \rangle$ the tuples τ where $\tau[x_i] = v_i$ [Lecoutre *et al.*, 2012].

A CSP is m-wise consistent if, every tuple in a relation can be extended to every combination of $m - 1$ other relations in a consistent manner [Gyssens, 1986; Janssen *et al.*, 1989]. Keeping with relational-consistency notations, Karakashian et al. denoted m-wise consistency by R($*$,m)C, and proposed a first algorithm for enforcing it [Karakashian *et al.*, 2010]. Their implementation finds an extension (i.e., support) for a tuple by conducting a backtrack search on the other $m-1$ relations, and removes the tuples that have no support. After all relations are filtered, they are projected onto the domains of the variables. Pairwise consistency (PWC) corresponds to m=2, R($*$,2)C≡PWC. Lecoutre et al. introduced the algorithm extended STR (eSTR) [Lecoutre *et al.*, 2013], which enforces PWC on a CSP using the STR mechanism [Ullmann, 2007]. eSTR maintains counters on the intersections of two constraints to determine if a tuple is pairwise consistent or not. In this chatper, we enforce PWC using the algorithm for R($*$,2)C [Karakashian *et al.*, 2010], and not eSTR, because it is prohibitively expensive to continuously maintain the counters of eSTR in a strategy where PWC is only selectively enforced.

B.2 Adaptive Parameterized Consistency

Balafrej et al. introduced *the distance to the end of value v_i for variable x_i* as:

$$\Delta(x_i, v_i) = \frac{|dom^o(x_i)| - rank(v_i, dom^o(x_i))}{|dom^o(x_i)|}$$

where $dom^o(x_i)$ is the original, unfiltered domain of x_i, and $rank(v_i, dom^o(x_i))$ is the position of v_i in the ordered set $dom^o(x_i)$ [Balafrej *et al.*, 2013]. In Figure B.1, borrowed from [Balafrej *et al.*, 2013], $\Delta(x_2, 1) = 0.75$, $\Delta(x_2, 2) = 0.50$, $\Delta(x_2, 3) = 0.25$, and $\Delta(x_2, 4) = 0.00$.

Further, for a given parameter p, they defined $\langle x_i, v_i\rangle$ to be *p-stable for AC* for c_{ij} where $scope(c_{ij}) = \{x_i, x_j\}$ if there exists an AC-support $\langle x_j, v_j\rangle$ with $\Delta(x_j, v_j) \geq p$ for c_{ij}. Figure B.1 illustrates an example for the constraint $x_1 \leq x_2$ with $p = 0.25$. $\langle x_1, 1\rangle$,$\langle x_1, 2\rangle$,$\langle x_1, 3\rangle$ are all 0.25-stable for AC for the constraint, but $\langle x_1, 4\rangle$ is not, because its only AC-support, $\langle x_2, 4\rangle$, has distance 0.

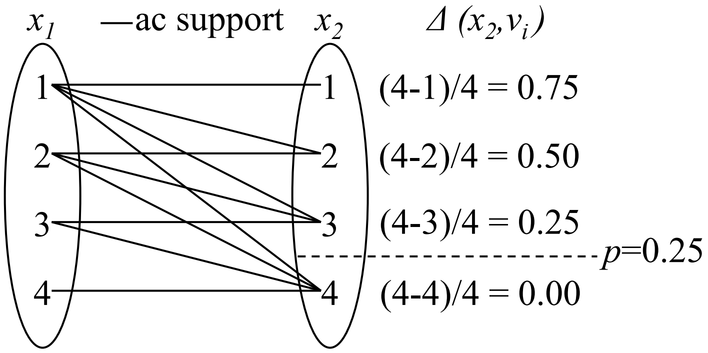


Figure B.1: The constraint $x_1 \leq x_2$. $\langle x_1, 4\rangle$ is not 0.25-stable for AC.

The *parameterized* strategy p-LC [Balafrej *et al.*, 2013] enforces, on each variable-value pair, either AC or some local consistency (LC) property strictly stronger than AC depending on the value of the parameter p. The idea is to enforce LC only on the variable-value pairs with few supports, approximated with the rank ($< p$) of the first found AC-support. We focus on the constraint-based version, pc-LC, where $\langle x_i, v_i\rangle$ is pc-LC if for *every* constraint $c_j \in cons(x_i)$, $\langle x_i, v_i\rangle$ is p-stable for AC on c_j or $\langle x_i, v_i\rangle$ is LC on c_j. In pc-LC, the value of p is given as input. In the *adaptive* version, *apc*-LC, it is dynamically determined for each constraint c_j using the *weight* of c_j, $w(c_j)$, which is the number of times c_j caused a domain wipe-out like in the variable-ordering heuristic dom/wdeg [Boussemart *et al.*, 2004]:

$$p(c_j) = \frac{w(c_j) - \min_{c_k \in \mathcal{C}}(w(c_k))}{\max_{c_k \in \mathcal{C}}(w(c_k)) - \min_{c_k \in \mathcal{C}}(w(c_k)) + 1}. \tag{B.1}$$

In [Balafrej *et al.*, 2013], *apc*-maxRPC was experimentally shown to outperform AC and maxRPC [Debruyne and Bessière, 1997a].

B.3 Modifying *apc*-LC for Non-Binary CSPs

For binary CSPs, p-stability for AC of $\langle x_i, v_i\rangle$ estimates how many supports are left for $\langle x_i, v_i\rangle$ in other constraints using the rank of the AC-support in the corresponding domain. This estimate should not directly applied to non-binary table constraints because the GAC-support of $\langle x_i, v_i\rangle$ is a tuple in a relation that is unsorted, which would make the estimate way too imprecise. Consider the example with $\langle x_i, v_i\rangle$ and a relation R_j of 100 tuples. Assume that the only tuple $\tau \in R_j$ supporting $\langle x_i, v_i\rangle$ appears at the top of the table of R_j. The estimate would indicate that there are many supports for $\langle x_i, v_i\rangle$ because there are 99 tuples that appear after it. However, in reality, $\langle x_i, v_i\rangle$ has a unique support. Below, we introduce p-stability for GAC, which counts the number of supports for each variable-value pair. Then, we introduce a mechanism to compute p-stability for GAC, and finally give an algorithm for enforcing *apc*-LC, which adaptively enforces STR or LC. In this chapter, we study R($*$,2)C as LC, and discuss the implementation of *apc*-R($*$,2)C.

B.3.1 p-stability for GAC

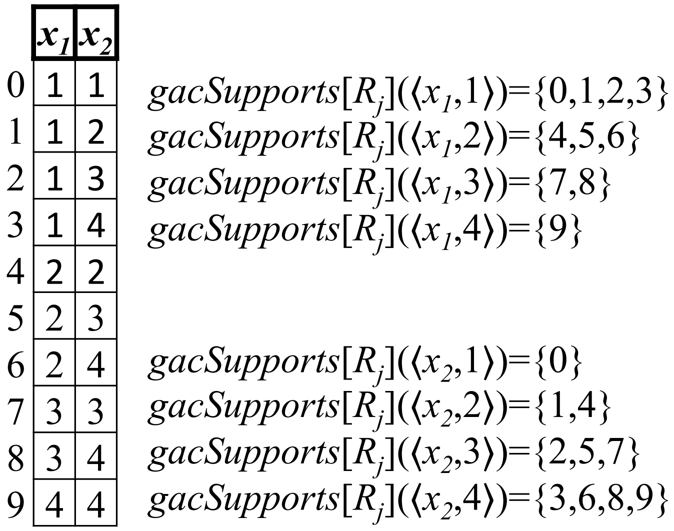
We say that $\langle x_i, v_i\rangle$ is *p-stable for GAC* if for every constraint $c_j \in cons(x_i)$,

$$\frac{|\sigma_{x_i=v_i}(R_j)|}{|R_j^o|} \geq p(c_j),$$

where $\sigma_{x_i=v_i}(R_j)$ selects the tuples in R_j where $\langle x_i, v_i\rangle$ appears, and R_j^o is the original, unfiltered relation. A CSP is p-stable for GAC if every variable-value pair is p-stable for GAC for every constraint that applies to it.

Figure B.2 gives the relation for the constraint $x_1 \leq x_2$. $\langle x_1, 1\rangle$ and $\langle x_1, 2\rangle$ are 0.25-stable for GAC. Indeed, $\sigma_{x_1=1}$ returns four rows $\{0, 1, 2, 3\}$ in the table, and $\langle x_1, 1\rangle$

is 0.25-stable: $\frac{4}{10} \geq 0.25$. Similarly, $\langle x_1, 2\rangle$ also is 0.25-stable: $\frac{3}{10} \geq 0.25$. $\langle x_1, 3\rangle$ and $\langle x_1, 4\rangle$ are not 0.25-stable, because $\frac{2}{10} \not\geq 0.25$ and $\frac{1}{10} \not\geq 0.25$. This example illustrates how, on binary constraints, and for a given p, p-stable for AC does not guarantee p-stable for GAC. (Recall that $\langle x_1, 3\rangle$ is 0.25-stable for AC in Figure B.1).



	x_1	x_2
0	1	1
1	1	2
2	1	3
3	1	4
4	2	2
5	2	3
6	2	4
7	3	3
8	3	4
9	4	4

$gacSupports[R_j](\langle x_1,1\rangle)=\{0,1,2,3\}$
$gacSupports[R_j](\langle x_1,2\rangle)=\{4,5,6\}$
$gacSupports[R_j](\langle x_1,3\rangle)=\{7,8\}$
$gacSupports[R_j](\langle x_1,4\rangle)=\{9\}$

$gacSupports[R_j](\langle x_2,1\rangle)=\{0\}$
$gacSupports[R_j](\langle x_2,2\rangle)=\{1,4\}$
$gacSupports[R_j](\langle x_2,3\rangle)=\{2,5,7\}$
$gacSupports[R_j](\langle x_2,4\rangle)=\{3,6,8,9\}$

Figure B.2: The relation of $x_1 \leq x_2$. $\langle x_1, 3\rangle$ and $\langle x_1, 4\rangle$ are not 0.25-stable for GAC.

B.3.2 Computing p-stability for GAC

For each constraint c_j, we introduce for every $\langle x_i, v_i\rangle$ a set of integers indicating the position of the tuples returned by $\sigma_{x_i=v_i}(R_j)$, which is similar to the data structure in GAC4 [Mohr and Masini, 1988]. We denote this table $gacSupports[R_j][\langle x_i, v_i\rangle]$. The check for p-stable can be verified by using $|gacSupports[R_j][\langle x_i, v_i\rangle]|$. Figure B.2, shows the $gacSupports[R_j]$ for the constraint $x_1 \leq x_2$. For each relation, the space complexity to store each $gacSupports[R_j]$ is $\mathcal{O}(k \cdot t)$, where k is the maximum constraint arity and t is the maximum number of tuples in a relation. The time complexity to generate $gacSupports[R_j]$ is $\mathcal{O}(k \cdot t)$, by iterating through every tuple.

B.3.3 Algorithm for Enforcing *apc*-LC

With the *gacSupports* data-structure, we can apply STR by verifying, for each constraint c_j, that every variable $x_i \in scope(c_j)$ and $v_i \in dom(x_i)$ has a non-zero $|gacSupports[R_j][\langle x_i, v_i\rangle]|$. Living-STR (Algorithm 20) does precisely this operation (ignoring Lines 4 and 5, which apply to the *apc*-LC operation introduced next). $past(\mathcal{P})$ denotes the variables of the CSP $\mathcal{P}$ already instantiated by search, and $\text{delTuples}(R_k, S, level)$ deletes all the tuples in the subset $S \subseteq R_k$, and marks their removal level at the level of search $level$. When deleting a tuple from the relation R_k, c_k's neighboring constraints, $neigh(c_k)$, should be re-queued to be processed with Living-STR. Initially, all constraints are in the queue. Living-STR is similar to STR3 in that it iterates over variable-value pairs rather than over tuples. However, it does not use as much book-keeping for optimizing the number of STR checks as STR3 [Lecoutre *et al.*, 2012]. Instead, Living-STR uses the same data structures as STR and STR2(+) to manage tuple deletions in a relation [Lecoutre, 2011; Ullmann, 2007].

Including Lines 4 and 5 in Algorithm 20 yields *apc*-LC, which adaptively applies LC. The adaptive level $p(c_j)$ is defined by Balafrej et al. [Balafrej *et al.*, 2013] and recalled in Equation (B.1). The local consistency technique used here is the implementation of R(∗,2)C [Karakashian *et al.*, 2010], *apc*-R(∗,2)C. Apply-R(∗,2)C (Algorithm 21) takes as input the list of tuples of a constraint on which R(∗,2)C must be enforced. SearchSupport$(R_i, \tau, \{R_j\})$ on Line 3 of Algorithm 21 searches for a support for the tuple $\tau \in R_i$, the pairwise check [Karakashian *et al.*, 2010].

Theoretical analysis: Let k be the maximum constraint arity, d the maximum domain size, and δ the maximum number of neighbors of a constraint. The time complexity of Algorithm 20 is $\mathcal{O}(k \cdot d)$. Algorithm 21 is $\mathcal{O}(\delta \cdot t^2)$ because it makes $\mathcal{O}(\delta \cdot t)$ calls

Algorithm 20: LIVING-STR(c_i): set of variables

```
Input: $c_j$: a constraint of $\mathcal{P}$
Output: Set of variables in $scope(c_j)$ whose domains have been modified
$X_{modified} \leftarrow \emptyset$
foreach $x_i \in scope(c_j) \mid x_i \notin past(\mathcal{P})$ do
    foreach $v_i \in dom(x_i)$ do
        if $|gacSupports[R_j](\langle x_i, v_i\rangle)| \neq 0$ and $\frac{|gacSupports[R_j](\langle x_i,v_i\rangle)|}{|R_j^o|} \ngeq p(c_j)$ then
            APPLY-LC($R_j, gacSupports[R_j](\langle x_i, v_i\rangle)$)
        if $|gacSupports[R_j](\langle x_i, v_i\rangle)| = 0$ then
            foreach $c_k \in cons(x_i)$ do
                delTuples($c_k, gacSupports[R_k](\langle x_i, v_i\rangle), |past(\mathcal{P})|$)
            $dom(x_i) \leftarrow dom(x_i) \setminus \{v_i\}$
            if $dom(x_i) = \emptyset$ then throw INCONSISTENCY

            $X_{modified} \leftarrow X_{modified} \cup \{x_i\}$
return $X_{modified}$
```

Algorithm 21: APPLY-R(∗,2)C(c_i, *tuples*)

```
Input: $c_i$: a constraint; tuples: a set of tuples from the constraint $c_i$
Output: The tuples are either R(∗,2)C or deleted
foreach $\tau \in tuples$ do
    foreach $c_j \in neigh(c_i)$ do
        if SEARCHSUPPORT($R_i, \tau, \{R_j\}$) returns inconsistent then
            delTuples($c_i, \{\tau\}, |past(\mathcal{P})|$)
```

to SEARCHSUPPORT, which is $\mathcal{O}(t)$ in our context. The correctness of Algorithms 20 and 21 can be shown in straightforward manner by contradiction.

B.4 Empirical Evaluations

The goal of our experimental analysis is to assess if *apc*-R(∗,2)C effectively selects when to apply STR and R(∗,2)C when used in a pre-processing step and in a real full lookahead strategy [Haralick and Elliott, 1980] during backtrack search to find the first solution to a CSP. In our experiments, we use the variable ordering dom/wdeg

[Boussemart *et al.*, 2004]. The experiments are conducted on the benchmarks of the CSP Solver Competition[1] with a time limit of two hours per instance and 8 GB of memory. Because STR and R(∗,2)C enforce the same level of consistency on binary CSPs [Bessière *et al.*, 2008], we focus our experiments on 21 non-binary benchmarks[2] consisting of 623 CSP instances. We chose these benchmarks because they are given in extension and at least one algorithm completed 5% of the instances in the benchmark.

Table B.1 summarizes the results in terms of number of instances solved. Importantly, *apc*-R(∗,2)C completes the largest number of instances (552). Considering the instances solved by all algorithms (485 instances), *apc*-R(∗,2)C has the smallest average and median CPU time. Row 3 indicates the number of instances STR solved but R(∗,2)C and *apc*-R(∗,2)C did not solve (18 and 11 instances, respectively), thus showing that *apc*-R(∗,2)C, although it may have enforced R(∗,2)C too often, outperformed R(∗,2)C and missed fewer instances than it (11 vs. 18). Row 4 exhibits similar results showing the number of instances that R(∗,2)C could solve, but that were missed by STR and *apc*-R(∗,2)C (64 and 6 instances, respectively). Here, *apc*-R(∗,2)C did not enforce R(∗,2)C often enough, but managed to outperform STR missing significantly fewer instances than STR (6 vs. 64).

Table B.1: Number of instances completed by the tested algorithms

		STR	R(∗,2)C	*apc*-R(∗,2)C
1	#instances completed by	504	550	**552**
2	#instances completed only by	10	5	0
3	#instances solved by STR, but missed by	0	18	11
4	#instances solved by R(∗,2)C, but missed by	64	0	6
5	#instances solved by *apc*-R(∗,2)C, but missed by	59	8	0
Average CPU time (sec.) over 458 instances		328.41	378.12	**313.31**
Median CPU time (sec.) over 458 instances		7.23	17.35	**7.21**

---

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

[2] aim-(50,100,200), allIntervalSeries, dag-rand, dubois, jnh(Sat/Unsat), lexVg, modifiedRenault, pret, rand-10-20-10, rand-3-20-20(-fcd), rand-8-20-5, ssa, travellingSalesman-20, travellingSalesman-25, ukVg, varDimacs, wordsVg

Table B.2 gives a finer analysis of the data, showing the number of completions and average and median CPU time per benchmark. Averages computed over only the instances completed by all techniques are shown in the column *All*. We split the table into four categories based on the *average* CPU time of *apc*-R(∗,2)C: *a*) *apc*-R(∗,2)C performs the best (5 benchmarks); *b*) *apc*-R(∗,2)C is competitive, performing between STR and R(∗,2)C (13 benchmarks); *c*) *apc*-R(∗,2)C performs the worst (2 benchmarks); and *d*) STR does not solve the benchmark but R(∗,2)C and *apc*-R(∗,2)C do (1 benchmark). The best average CPU time appears in bold face in the corresponding column. The median CPU time of *apc*-R(∗,2)C is bold faced when its rank differs from that of the average CPU time (on which the four categorized are based).

Table B.2: Results of the experiments per benchmark, organized in four categories

Benchmark	#Instances	#Completed				Average CPU time (sec)			Median CPU time (sec)		
		STR	R(∗,2)C	*apc*-R(∗,2)C	All	STR	R(∗,2)C	*apc*-R(∗,2)C	STR	R(∗,2)C	*apc*-R(∗,2)C
a) *apc*-R(∗,2)C is the best											
aim-50	24	24	24	24	24	**0.04**	0.07	**0.04**	0.02	0.04	**0.03**
allIntervalSeries	25	22	22	22	22	7.09	141.85	**6.00**	0.13	0.31	0.12
jnhSat	16	16	16	16	16	13.07	357.66	**11.74**	8.15	142.24	7.21
modifiedRenault	50	50	50	50	50	6.39	11.17	**6.29**	7.24	8.79	6.98
rand-3-20-20	50	31	43	41	31	1,666.10	939.88	**932.77**	1,211.50	822.54	811.74
b) *apc*-R(∗,2)C is competitive											
aim-100	24	24	24	24	24	0.38	**0.26**	0.41	0.18	0.25	**0.16**
aim-200	24	22	24	24	22	414.48	**6.52**	286.27	2.39	1.37	**2.60**
jnhUnsat	34	34	34	34	34	**13.61**	294.77	13.95	10.74	153.50	**9.78**
lexVg	63	63	63	63	63	**69.81**	341.87	338.74	0.50	1.38	0.89
pret	8	4	4	4	4	**117.89**	347.03	136.04	115.81	354.82	145.70
rand-3-20-20-fcd	50	39	48	47	39	928.06	**546.84**	615.23	501.30	422.24	464.00
rand-8-20-5	20	9	20	20	9	2,564.94	**355.57**	372.76	1,987.35	314.26	**261.68**
rand-10-20-10	20	12	12	12	12	6.72	**1.67**	2.76	6.40	1.66	2.75
ssa	8	6	5	6	5	**64.60**	100.64	69.59	1.51	1.60	1.58
TSP-25	15	13	10	13	10	**232.38**	1,072.72	743.33	69.00	211.41	131.69
ukVg	65	37	31	34	31	**166.82**	796.90	421.35	36.29	54.65	**30.39**
varDimacs	9	6	6	6	6	**89.23**	587.55	319.20	1.56	6.43	2.94
wordsVg	65	65	58	58	58	**119.76**	532.05	400.22	0.39	0.95	0.59
c) *apc*-R(∗,2)C is the worst											
dubois	13	7	8	6	6	1,000.54	**451.91**	1,456.01	552.13	255.25	779.57
TSP-20	15	15	15	15	15	**101.20**	318.37	335.13	23.32	61.55	**46.34**
d) Not solved by STR											
dag-rand	25	0	25	25	0	-	**123.70**	149.64	-	124.47	151.33

On TSP-20, *apc*-R($*$,2)C ranks bottom on average CPU time but between STR and R($*$,2)C on median CPU time. On aim-100, jnhUnsat, rand-8-20-5, and ukVg, *apc*-R($*$,2)C is between STR and R($*$,2)C for average CPU time, but best for median CPU time.

Table B.3 shows the average number of STR and R($*$,2)C checks that *apc*-R($*$,2)C performs per benchmark. In allIntervalSeries, *no* calls are made to R($*$,2)C because the instance is solved backtrack free with STR alone. For *apc*-LC, *no call to LC is done during pre-processing* because the weights of all the constraints are set to 1 (giving $p(c_j) = 0$ for all $c_j \in \mathcal{C}$) and updated only during search. For dag-rand, there is a smaller number of R($*$,2)C calls than STR calls (21,870 vs. 359,248). However, those few calls allow us to solve *all* the instances of this benchmark whereas STR alone could not solve *any* instance. This result is a glowing testimony of the ability of *apc*-R($*$,2)C to apply the appropriate level of consistency where needed.

Table B.3: Number of calls to STR and R($*$,2)C by benchmark

Benchmark	STR checks	R($*$,2)C checks
a) apc-R($*$,2)C is the best		
aim-50	456,823	39,491
allIntervalSeries	38,281,694	0
jnhSat	22,119,135	599,080
modifiedRenault	4,618,778	601,641
rand-3-20-20	489,441,126	3,480,216,943
c) apc-R($*$,2)C is the worst		
dubois	3,343,830,604	4,668,288
TSP-20	622,949,698	991,590,957
d) Not solved by STR		
dag-rand	359,248	21,870

Benchmark	STR checks	R($*$,2)C checks
b) apc-R($*$,2)C is competitive		
aim-100	7,731,585	894,353
aim-200	1,160,334,482	163,177,907
jnhUnsat	51,688,166	1,918,781
lexVg	564,010,457	2,180,503,026
pret	422,987,946	13,973,748
rand-3-20-20-fcd	455,664,100	2,956,467,994
rand-8-20-5	77,470,561	184,764,543
rand-10-20-10	72,608	3,972
ssa	156,631,370	11,689,961
TSP-25	2,903,953,315	3,947,391,769
ukVg	341,565,892	1,002,334,753
varDimacs	720,843,958	84,123,204
wordsVg	514,840,737	2,052,367,934

Summary

In this chapter, we extend the notion of *p*-stability for AC to GAC, and provide a mechanism for computing it. We give an algorithm for enforcing *apc*-R($*$,2)C on non-binary table constraints, which adaptively enforces GAC and R($*$,2)C. We validate our approach on benchmark problems. Future work is to investigate other adaptive criteria for selecting the level of consistency to apply, in particular one that operates during both pre-processing and search. To apply our approach to constraints defined in intension and other global constraints, we could use techniques that approximate the number of solutions in those constraints [Pesant *et al.*, 2012].

Appendix C

Witness-Based Search for Solution Counting

Counting the exact number of solutions of a Constraint Satisfaction Problem (CSP) is an important but difficult task. To overcome this difficulty, the techniques proposed in the literature organize the search process along a tree decomposition of the CSP, where all the extensions of a given partial solution over different branches of the tree are first independently counted in each branch before their numbers can be multiplied. We observe that this count is zero when any of the branches has no solution. We propose witness-based search, which first ensures the existence of a solution (i.e., witness) in each branch before starting the counting. We empirically establish the benefits of our technique in the context of the BTD and AND/OR search graphs.

C.1 Introduction

Counting the number of solutions of a Constraint Satisfaction Problem (CSP), an important task in verification and automated reasoning, is known to be #P-complete

[Valiant, 1979]. Current techniques for solving this problem exploit some *tree structure* of the constraint network of the CSP in order to reduce the search and counting efforts [Dechter and Pearl, 1988; Gogate and Dechter, 2008; Favier *et al.*, 2009].

Indeed, in a tree-structured problem, the number of solutions at any node in the tree is computed by simple algebraic operations (i.e., summation and product) from the number of solutions of the children of the node and information at the node itself, following a pre-order traversal. In a non-parallel implementation, all the solutions in one branch of the tree are counted before the solutions in another branch with the same parent. In case the latter branch has no solution, the effort spent counting the solutions in the first branch are wasted. We propose to first *find* a witness solution in every branch of a given node in the tree before proceeding to *counting* the number of solutions in any given branch. We call this scheme *witness-based search*.

Further, tree-structured methods typically and heavily exploit a caching mechanism. This mechanism maintains, at some nodes of the search space, results that were derived during search in order to reduce the amount of repetitive and redundant work done during search. The information cached includes (portions of) partial solutions that yielded inconsistencies (i.e., nogoods) and also those that yielded solutions (i.e., goods) along with the count of solutions found.

We apply witness-based search to two solution-counting methods, namely, the Backtrack Search with Tree Decomposition (BTD) [Jégou and Terrioux, 2003] and the AND/OR search tree [Dechter and Mateescu, 2004]. Our empirical evaluations show a reduction of the search effort, and, importantly, the space used for caching, which is a major bottleneck in those techniques.

This chapter is structured as follows. Section C.2 recalls main concepts and definitions. Section C.3 discusses solution-counting methods based on tree structures. Section C.4 describes and discusses witness-based solution counting. Section C.5

describes our experiments.

C.2 Main Definitions

We first summarize the main concepts and definitions used.

C.2.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) is defined by $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a set of domains, and $\mathcal{C}$ is a set of constraints. Each variable in $\mathcal{X}$ has a finite domain in $\mathcal{D}$, and is constrained by a subset of the constraints in $\mathcal{C}$. Each constraint $C_i \in \mathcal{C}$ is defined by a relation R_i specified over the *scope* of the constraint, $\mathit{scope}(C_i)$, which are the variables to which the constraint applies, as a subset of the Cartesian product of the domains of those variables. A tuple $t_i{\in}R_i$ is a combination of values for the variables in the scope of the constraint that is either allowed (i.e., support) or forbidden (i.e., conflict). A solution to the CSP is an assignment to each variable of a value taken from its domain such that all the constraints are satisfied. In general, finding a solution to a CSP is NP-complete, and counting its number of solutions is #P-complete.

Backtrack search is a sound and complete algorithm commonly used to solve CSPs. To improve the performance of search and reduce the severity of the combinatorial explosion, we enforce a given local consistency level. One common such property is Generalized Arc Consistency (GAC). A CSP is GAC iff for every constraint, any value in the domain of any variable in the scope of the constraint can be extended to a tuple satisfying the constraint.

Several graphical representations of a CSP exist. In the *hypergraph*, the vertices represent the variables of the CSP, and the hyperedges represent the scopes of the constraints (see Figure C.1). In the *primal graph*, the vertices represent the CSP variables, and the edges connect every two variables that appear in the scope of some constraint (see Figure C.2).

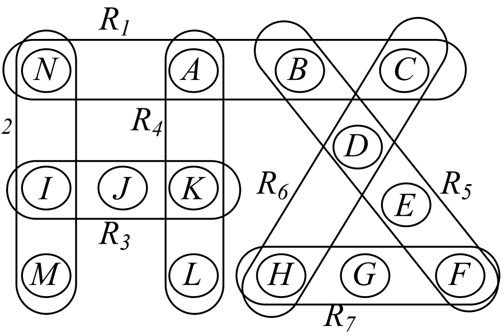


Figure C.1: A hypergraph

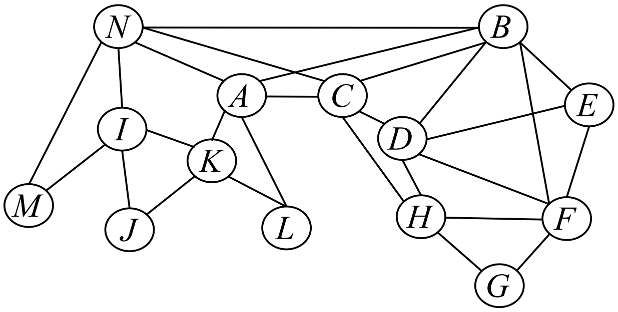


Figure C.2: The primal graph

C.2.2 Backtrack Search with Tree Decomposition

A *tree decomposition* of a CSP is a tree embedding of its constraint network. The tree nodes are *clusters* of variables and constraints from the CSP. The set of variables of a cluster cl is denoted $\chi(cl) \subseteq \mathcal{X}$, and the set of constraints $\psi(cl) \subseteq \mathcal{C}$. A tree decomposition must satisfy two conditions:

1. Each constraint appears in at least one cluster and the variables in its scope must appear in this cluster; and

2. For every variable, the clusters where the variable appears induce a connected subtree.

Many techniques for generating a tree decomposition of a CSP exist [Dechter and Pearl, 1989; Jeavons *et al.*, 1994; Gottlob *et al.*, 2000]. We use here the tree-clustering technique [Dechter and Pearl, 1989]. *First*, we triangulate the primal graph of the

CSP using the min-fill heuristic [Kjærulff, 1990]. *Second*, using the perfect elimination ordering given by the MaxCardinality algorithm [Tarjan and Yannakakis, 1984], we identify the maximal cliques in the resulting chordal graph using the MaxCliques algorithm [Golumbic, 1980], and use the identified maximal cliques to form the clusters of the tree decomposition. Figure C.3 shows a triangulated primal graph of the example in Figure C.1. The dotted edges (B,H) and (A,I) in Figure C.3 are fill-in edges generated by the triangulation algorithm. The ten maximal cliques of the triangulated graph are highlighted with 'blobs.' *Third*, we build the tree by connecting the clusters using the JoinTree algorithm [Dechter, 2003a]. While any

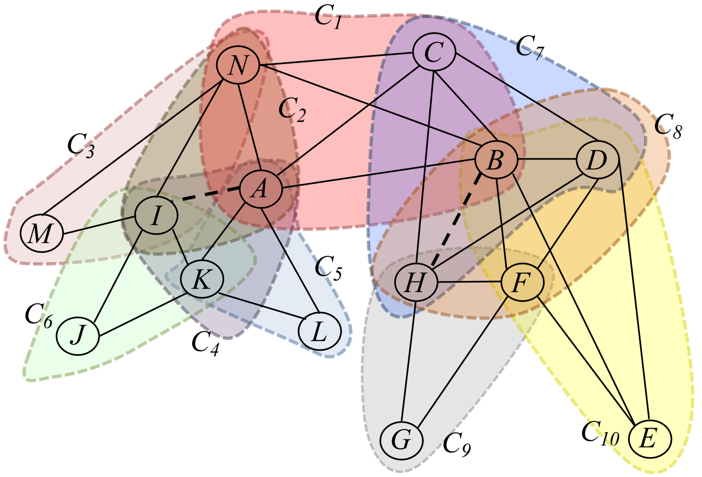


Figure C.3: Triangulated primal graph and its maximal cliques

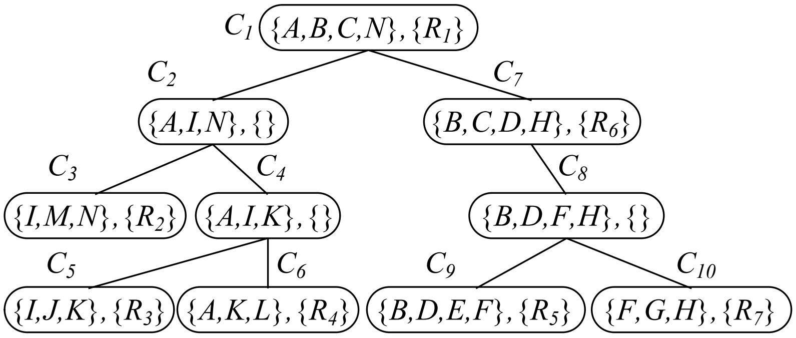


Figure C.4: A tree decomposition of the CSP in Figure C.1

cluster can be chosen as the root of the tree, we choose the cluster that minimizes the longest chain from the root to a leaf. Figure C.4 shows the tree after connecting the maximal cliques of Figure C.3. *Finally*, we determine the variables and constraints of each cluster as follows: *a*) The variables of a cluster cl, $\chi(cl)$, are the variables in the maximal clique that yields the cluster; and *b*) The constraints of a cluster cl, $\psi(cl)$, are all the constraints R_i, such that $scope(R_i) \subseteq \chi(cl)$. Figure C.4 shows a tree decomposition for the example of Figure C.1. Note that we may end up with clusters with no constraints (e.g., C_2, C_4 and C_8). A *separator* of two adjacent clusters is the set of variables that are associated with both clusters.

C.2.3 AND/OR Tree Search

AND/OR tree search was proposed by Dechter [2004] as a generalization of search in graphical models. AND/OR tree search exploits (in)dependencies in the model to exponentially reduce the search effort, binding it exponentially by, instead of the number of variables, the depth of a *pseudo-tree* [Freuder and Quinn, 1987], which is a tree spanning of the model. Dechter also extended the AND/OR search space from a tree to a *graph*, further reducing the time effort albeit at the cost of increased memory space [2004]. The detailed definitions and characterizations are accessible in the original papers; below we illustrate this process with a simple example.

Consider a CSP with the constraint graph shown in Figure C.5. The domain of the variable Y is $\{2, 3\}$. The domains of W, X, Z, T, R are $\{1, 2\}$. The constraints are as follows: $W = Z$, $W = R$, $W \geq X$, $0 \leq T - X \leq 1$, and $X < Y$. The constraint between Y and T forbids only the tuple $\langle(Y, 2), (T, 1)\rangle$. Similarly, the constraint between Y and R forbids only the tuple $\langle(Y, 2), (R, 1)\rangle)$. Figure C.6 gives a pseudo-tree of this CSP where the dependencies between variables are shown as the tree edges

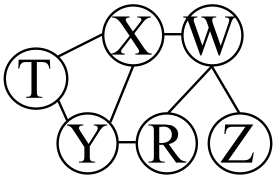


Figure C.5: A constraint graph

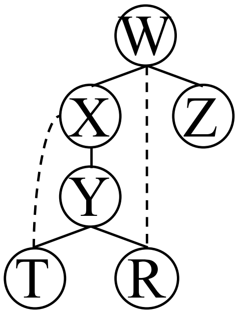


Figure C.6: A pseudo-tree of the example from Figure C.5

(full lines) and back-edges (dotted lines). Figure C.7 shows the AND/OR search tree of the example in Figure C.5 using the pseudo-tree of Figure C.6. An AND/OR search tree alternates between OR nodes (variables) and AND nodes (variable as-

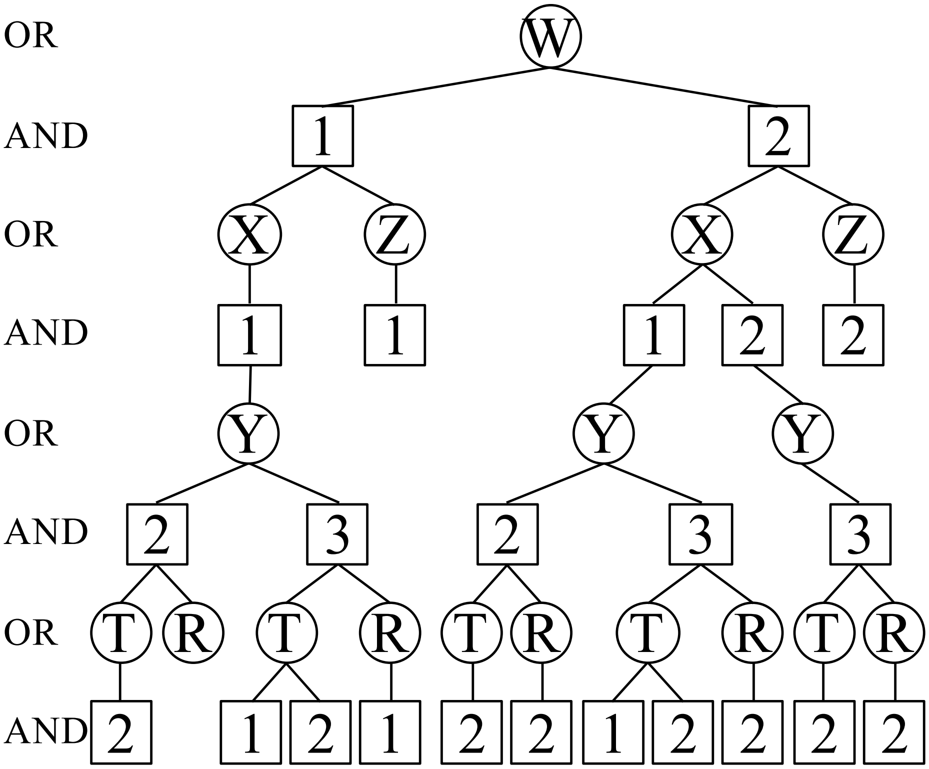


Figure C.7: An AND/OR search tree of the example from Figure C.5

signments). The structure of the AND/OR search tree is based on the pseudo-tree. The root of the AND/OR search tree is an OR node for the variable at the root of the pseudo-tree. The children of an OR node are AND nodes corresponding to the value assignments of the variable of the OR node. The children of an AND node are OR nodes, corresponding to the variables that are the children of the AND node's variable in the pseudo-tree.

The *parents* of an OR node V are the ancestors of V in the pseudo-tree that are connected in the constraint graph to V or to descendants of V. The *parent-separator* of an OR node V (or an AND node $\langle V, v\rangle$) is the set containing V and its ancestors in the pseudo-tree that are connected in the original graph to descendants of V. The *context of an* AND *node* is the assignments of the variables in the node's parent-separator. The *context of an* OR *node* is the assignments of the variables in the node's parents. Two nodes can be merged together if their context is the same, thus yielding a search graph. Figure C.8 shows the AND/OR search graph of our example using OR context-merging. Note that we could merge nodes on *both* the OR context and AND context; however, merging with one context makes the other unnecessary [Dechter and Mateescu, 2006].

C.3 Tree-Based Solution Counting

Below, we discuss solution-counting methods and provide a pseudo-code that operates on binary tree-structured CSPs, the BTD, and AND/OR search graphs. In Section C.4, we modify this pseudo-code to incorporate our witness mechanism. Our pseudo-code is specified recursively for readability, but our implementation is iterative. Further, the pseudo-code relies on back-checking for extending consistent partial solutions, whereas our implementation uses look-ahead.

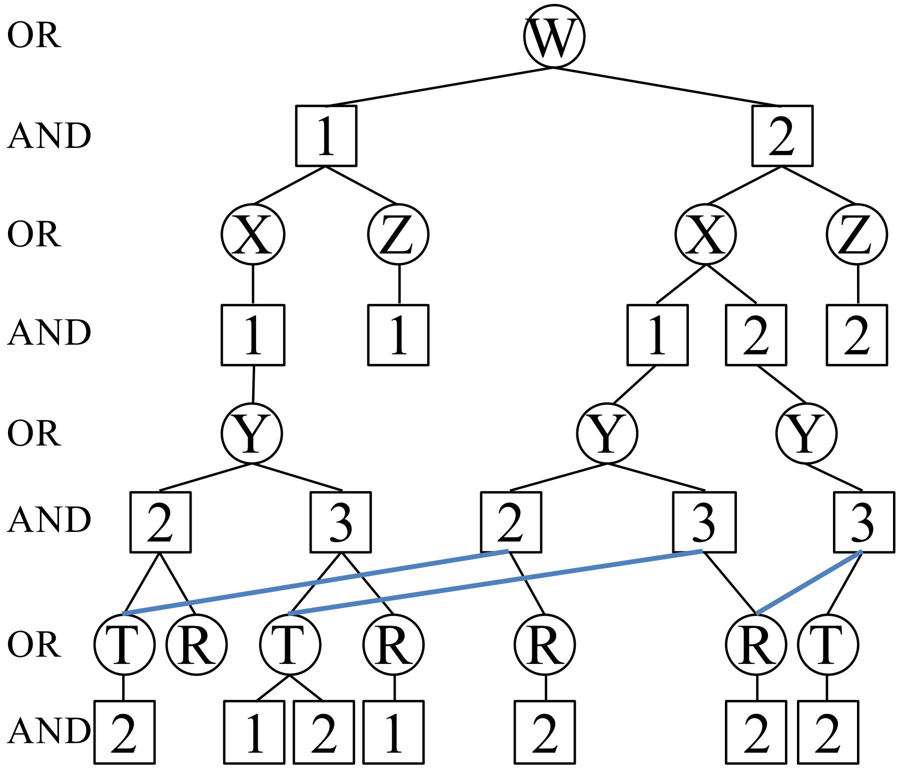


Figure C.8: The AND/OR search graph by merging OR contexts

C.3.1 Solution Counting in a Tree-Structured Binary CSP

Dechter and Pearl [1988] noted that the number of solutions in a tree-structured binary CSP can be computed in $O(nd^2)$ where n is the number of variables and d the maximum domain size. It computes the number of solutions of a given CSP variable from the number of solutions of its children in the tree. In summary,

1. the number of solutions rooted at a given variable in the tree-structured CSP is the summation of the number of solutions 'rooted' at each value in the domain of the variable; and,

2. the number of solutions at a given value of the domain is the product of the numbers of solutions of the value's consistent extensions in each of the children

of the variable.

We wrote Algorithm 22 to loosely accommodate all three solution methods discussed in this section. The algorithm is started by running #Sols(*root*,$\emptyset$), where *root* is the root of the tree. $\mathit{SolCache}(\mathit{child}, \mathcal{A})$ is the cache of a node given a partial assignment $\mathcal{A}$, and stores, when *bound*, the number of solutions rooted at the node. n_{total} stores the number of solutions at the *root*, and n_c stores the number of solutions rooted at the assignment $\mathit{root} \leftarrow v$. Whenever $n_c = 0$ within the loop of Lines 4–11, we exit the loop. This test is omitted for readability. The original procedure of Dechter and Pearl [1988] is easily obtained by ignoring the cache (Lines 5, 6, 7, 9, and 10).

Algorithm 22: #Sols(*root*,$\mathcal{A}$)

```
   Input: root of a tree structure of a CSP
          $\mathcal{A}$: A current partial solution
   Output: Number of solutions at root
 1 $n_{total} \leftarrow 0$
 2 foreach $v \in Domain(root)$ s.t. $v$ is consistent with $\mathcal{A}$ do
 3     $n_c \leftarrow 1$; $\mathcal{A}_{cur} \leftarrow \mathcal{A} \cup \{root \leftarrow v\}$
 4     foreach $child \in Children(root)$ do
 5         if $SolCache(child, \mathcal{A}_{cur})$ is bound then
 6             $cache \leftarrow SolCache(child, \mathcal{A}_{cur})$
 7         else
 8             $cache \leftarrow$ #Sols($child$, $\mathcal{A}_{cur}$)
 9             $SolCache(child, \mathcal{A}_{cur}) \leftarrow cache$
10             Cache good, no-good
11         $n_c \leftarrow n_c \times cache$
12     $n_{total} \leftarrow n_{total} + n_c$
13 return $n_{total}$
```

C.3.2 Solution Counting in the BTD

In the case of the BTD, Algorithm 22 operates on a tree decomposition of the CSP. Line 2 is called on the last unassigned variable in the root cluster as *root*. The child in Line 4 is the first unassigned variable in a child cluster. Before the recursive call in Line 8 is done on the last unassigned variable in the child cluster, we must first consistently extend the partial solution over the unassigned variables in the child cluster except for one variable.

When search succeeds, the BTD caches the instantiation of the variables at the separators as a 'good' along with the number of solutions rooted at this instantiation. Otherwise, the instantiations at the separator is cached as a 'no-good.'

C.3.3 Solution Counting in an AND/OR Search Tree

In the case of an AND/OR search tree, Algorithm 22 operates on the pseudo-tree. To count the solutions, the AND nodes multiply the numbers of solutions of their children (leaf AND nodes are considered to have one solution); OR nodes add the number of solutions of their children (leaf OR nodes are considered to have 0 solutions).

The cached information is similar to that cached by the BTD, except that it is for the instantiations of the variables in the contexts (not at the separators). The performance of this method is improved by the detection of dead-caches, which are caches that will never be hit [Darwiche, 2001; Marinescu and Dechter, 2006], and, thus, need not be recorded. In the presence of a dead-cache, the assignment of $\mathit{SolCache}(\mathit{child}, \mathcal{A})$ in Line 9 is not executed, and goods/no-goods are not stored in Line 10. Note that, the space needed for caching is a major bottleneck in tree-based solution-counting methods. Other techniques for dealing with this bottleneck exist (e.g., naive-caching and adaptive-caching [Marinescu and Dechter, 2006]) and are orthogonal to our approach.

C.4 Solution Counting in Witness-Based Search

The idea of our witness-based search is to refrain from counting solutions in any branch off a node in a tree structure before ensuring that the current partial solution at the node[1] can be consistently extended over the variables in *each* branch off the node. Indeed, if the solution fails to extend consistently over the variables of a single branch, then all the counting effort in the branches is wasted by multiplication by 0.

C.4.1 A Generic Pseudo-Code for Witness-Based Search

We specify witness-based search in the generic pseudo-code of Algorithm 23, and claim that it is applicable to any tree-based solution-counting method. Our technique interacts too tightly with the solution-counting strategies for it to be implemented as a separate component of a Constraint Solver. Indeed, the caching information stored by the various solution-counting strategies depend on the strategy itself. Based on our experience with two such strategies (i.e., BTD and AND/OR tree search), we found that the code of such strategies must be directly modified to *incorporate* witness-based search.

Algorithm 23 differs from Algorithm 22 by the use of a switch variable *mode*, which takes one of two values *sat* or *count* to determine whether search should check for satisfiability (i.e., find a witness) or do solution counting, respectively. The algorithm is started by running W#Sols(*root*,∅,*count*), where *root* is the root of the tree. In Line 2, the algorithm examines all the children, either finding a witness in the cache (Line 5) or doing the search to find a witness (Line 8). If *mode*=*sat*, then 1 is

---

[1] An AND node in the case of an AND/OR search tree.

Algorithm 23: W#Sols($root$,$\mathcal{A}$,$mode$)

```
   Input: root of a tree structure of a CSP
          $\mathcal{A}$: A current partial solution
          mode: Either sat for satisfiability or count for solution counting
   Output: If mode=count, number of solutions at root. Otherwise
           (mode=sat), 1 if a witness was found, 0 otherwise
1  $n_{total} \leftarrow 0$
2  foreach $v \in Domain(root)$ s.t. $v$ is consistent with $\mathcal{A}$ do
3      $n_c \leftarrow 1$; $\mathcal{A}_{cur} \leftarrow \mathcal{A} \cup \{root \leftarrow v\}$
4      foreach $child \in Children(root)$ do
5          if $SolWitnessCache(child, \mathcal{A}_{cur})$ is bound then
6              $cache \leftarrow SolWitnessCache(child, \mathcal{A}_{cur})$
7          else
8              $cache \leftarrow$ W#Sols($child$, $\mathcal{A}_{cur}$, $sat$)
9              $SolWitnessCache(child, \mathcal{A}_{cur}) \leftarrow cache$
10             Cache good, no-good
11         $n_c \leftarrow n_c \times cache$
12     if $mode$=$sat$ and $n_c > 0$ then return 1
13
14     if $mode$=$count$ and $n_c > 0$ then
15         $n_c \leftarrow 1$
16         foreach $child \in Children(root)$ do
17             if $SolCache(child, \mathcal{A}_{cur})$ is bound then
18                 $cache \leftarrow SolCache(child, \mathcal{A}_{cur})$
19             else
20                 $cache \leftarrow$ W#Sols($child$, $\mathcal{A}_{cur}$, $count$)
21                 $SolCache(child, \mathcal{A}_{cur}) \leftarrow cache$
22                 Cache good
23             $n_c \leftarrow n_c \times cache$
24         $n_{total} \leftarrow n_{total} + n_c$
25 return $n_{total}$
```

returned (Line 12). If *mode=count* and a witness is found, the algorithm proceeds to counting the number of solutions (Lines 14 to 23). Comparing Line 10 and Line 22 only goods are cached when *mode=count* because satisfiability is guaranteed by the witness mechanism.

C.4.2 Analysis of Witness-Based Search

In order to save on the search effort, the implementation of the algorithm should preserve the state of the search space in a branch where a witness is found so that, when the same branch is revisited again to count the remaining solutions, the effort to find the first solution is not repeated and the search can proceed from the witness. Below, we discuss two implementation strategies for handling the state of the search space where a witness solution was found. The first strategy does not always preserve the state of this space, whereas the second does.

In the first implementation strategy, after finding a witness in a branch br_i, we maintain the instantiations of the variables in this branch (i.e., freeze the search space in br_i) while checking on the other branches (which are independent of br_i). Thus, the recursive call in Line 20 to count solutions in br_i can continue from the current (frozen) state of the search. However, when backtracking occurs in the search above br_i, the variables in br_i and up to the backtrack level are uninstantiated (i.e., the search space in br_i is reset). When search resumes, and if the current path 'conditions' br_i in the same way as it did earlier,[2] we know, because of the stored good, that br_i has a witness. However, the state of the search space in br_i was reset because of backtracking. Thus, solution counting will have to restart from scratch. The advantage of this implementation is that it does not add to the memory space requirements. Its disadvantage is that, upon backtracking, the effort to find this first-solution has to be repeated.

The second implementation strategy is similar to the first, except that the caching is enhanced to also store the variable-value assignments of the witness (i.e., the first solution in br_i). Thus, upon backtracking, the state of the search space in br_i is restored and search can continue from that state when counting the number of solu-

---

[2] Determined by the instantiations of the variables in the separators/contexts.

tions in br_i (Line 20). The advantage of this strategy is that the effort to find the first-solution need not be repeated. However, the storage size for each cached good is increased linearly in the number of the variables in the branch.

While the first implementation strategy cannot guarantee that witness-based search does not increase the number of nodes visited by search, the second strategy does. We implemented both strategies: the first for the BTD, and the second for AND/OR tree search. We found them both to be advantageous on the tested instances despite the occasional and slight increase in the number of nodes visited by the witness-based BTD.

C.5 Empirical Evaluations

Our experiments assess the improvement brought about by the witness mechanism on solution-counting methods. We show that adding witness to both the BTD and AND/OR tree search results in significant improvements of both time and space on both unsatisfiable and satisfiable CSP instances.

C.5.1 Experimental Set-Up

We integrate GAC (GAC2001 [Bessière *et al.*, 2005]) in all our search algorithms as a real full look-ahead strategy. We find the pseudo-tree using the technique described by Bayardo and Mirankar [1996]. We instantiate the variables in the order of the pseudo-tree.

The experiments are conducted on the benchmarks of the CSP Solver Competition[3] with a time limit of two hours per instance and 8 GB of memory. We provide plenty of time and memory, to the extent possible, to avoid tainting our experiments

---

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

with censored data. We use benchmarks[4] that are difficult for BTD and AND/OR tree search to illustrate the advantage of using the witness technique in a challenging context. We split our analysis on the 479 unsatisfiable and 200 satisfiable instances tested.

It is not our goal to compare the performances of BTD with AND/OR tree search, but to evaluate the improvement brought about by witness-based search on each of them. For each of the two solution-counting methods, we focus our analysis on instances that were completed by search with and without the witness technique. Of the original 679 instances, the number of those instances is 308 for BTD and 239 for AND/OR search tree.[5] Further, we ignore the instances where the performance did not change in terms of nodes visited (on those instances the CPU time difference was within *less than 0.1%* and they used the same caching space). We end up with 106 instances for BTD and 95 instances for AND/OR search tree.[6] We analyze the performance by reporting the following measurements: *a*) the number of nodes visited, *b*) the CPU run time in seconds, and *c*) the space requirement in terms of number of goods and no-goods stored. The information about the witness needed to restore the state of the search space is included in the goods measurement. We show that witness-based search is advantageous by all three measurements.

C.5.2 Comparing Witness-BTD with BTD

In Tables C.1-C.3, we abbreviate Witness-BTD as W-BTD.

---

[4] aim-(50, 100, 200), composed-(25-10-20, 25-1-2, 25-1-25, 25-1-40, 25-1-80, 75-1-2, 75-1-25, 75-1-40, 75-1-80), dag-rand, dubois, graphColoring-(hos, mug, register-mulsol, register-zeroin, sgb-book, sgb-games, sgb-miles, sgb-queen), hanoi, modifiedRenault, QCP-15, rand-(10-20-10, 8-20-5), rlfap(GraphsMod, Scens11, ScensMod), ssa, and tightness0.9

[5] For BTD: 197 unsatisfiable, 111 satisfiable. For AND/OR search tree: 155 unsatisfiable, 84 satisfiable.

[6] For BTD: 69 unsatisfiable, 37 satisfiable. For AND/OR search tree: 59 unsatisfiable, 36 satisfiable.

Number of nodes visited: Table C.1 shows the number of instances that a given technique visits fewer nodes than the other, and the average number of nodes visited by each algorithm. Note that BTD *never* outperforms Witness-BTD on unsatisfiable instances. On satisfiable instances, Witness-BTD wins more often than BTD (29 instances). However, there are instances where BTD visits fewer nodes than Witness-BTD (8 instances). The reason is because the implementation of Witness-BTD does not restore the search space for cached witnesses, but instead searches again for the first solution, as discussed in Section C.4.2. Witness-BTD clearly outperforms BTD on unsatisfiable instances, showing substantial savings of not searching partial solutions that never participate in a global solution. On satisfiable instances, the difference is not as significant, albeit it shows an improvement. Notice, that although some search effort was wasted in our implementation of Witness-BTD (BTD visited fewer nodes on 8 instances than Witness-BTD), Witness-BTD still always saves on average on the number of nodes visited.

Table C.1: Number of instances with fewest #NV, and average #NV

	BTD	W-BTD
Fewest #NV		
UNSAT (69)	0	**69**
SAT (37)	8	**29**
Average #NV		
UNSAT (69)	1,431,275.77	**616,502.46**
SAT (37)	8,235,685.41	**8,166,271.57**

Run time: The savings in the number of nodes visited match those exhibited by the CPU time. Table C.2 reports the number of instances on which a given algorithm completed fastest within the CPU clock-resolution of 100 ms (thus, with occasional ties), and the average CPU time. On unsatisfiable instances, Witness-BTD solves more instances fastest than BTD and has a smaller average CPU time. On satisfiable

Table C.2: #Instances completed fastest and average time

	#Fastest		Avg. time (sec.)	
	BTD	W-BTD	BTD	W-BTD
UNSAT (69)	21	**55**	135.28	**110.01**
SAT (37)	**21**	17	724.51	**723.97**

instances, BTD is fastest on more instances than Witness-BTD (21 vs. 17 instances). However, the average CPU time is slightly less for the Witness-BTD. Thus, Witness-BTD yields savings (on unsatisfiable instances) while causing no significant overhead (on satisfiable instances).

Space requirements: Table C.3 gives the average number of stored goods and no-goods. Notice that the number of no-goods for Witness-BTD and BTD are almost identical, which is to be expected given that Witness-BTD finds the same no-goods, only earlier. *However, the number of goods stored is significantly reduced by* Witness-BTD. This fact illustrates how Witness-BTD avoids storing partial solutions that cannot be completed to global solutions, which is exactly our intended design.

Table C.3: Average number of goods and no-goods stored

	BTD	W-BTD
Average #no-goods		
UNSAT(69)	43,675.77	**43,675.74**
SAT(37)	449,633.21	**449,516.29**
Average #goods		
UNSAT(69)	24,104.52	**10,611.10**
SAT(37)	160,025.63	**148,739.58**

In summary, Witness-BTD achieves its goal: it saves on the number of nodes visited, time, and space, and never yields any overheads. It is a safe and robust strategy to implement in all circumstances, and clearly improves BTD. Therefore, it can be safely applied at all times.

C.5.3 Comparing Witness-AND/OR with AND/OR Tree Search

In Tables C.4-C.6, we abbreviate AND/OR tree search as AO and witness-AND/OR tree-search as W-A/O.

Number of nodes visited: As stated in Section C.4.2, Witness-AND/OR tree search is guaranteed to never visit more nodes than AND/OR tree search does. The variable-value assignments of the witness are cached so that the state of the search space can be restored to allow solution counting to resume from the witness. Table C.4 gives the average number of nodes visited by each strategy and shows a large reduction on both satisfiable and unsatisfiable instances.

Table C.4: Average #NV

	A/O	W-A/O
UNSAT (59)	580,762.02	**537,552.56**
SAT (36)	24,314,616.44	**19,521,667.08**

Run time: Once again, the reduction of the nodes visited directly translates into CPU time savings. Table C.5 shows the number of instances on which a given algorithm completed the fastest (within the CPU clock-resolution) and the average CPU time. AND/OR tree search did complete a few instances fastest (19 unsatisfiable and 10 satisfiable). However, note that the 19 unsatisfiable instances that AND/OR tree

Table C.5: #Instances completed fastest and average time

	#Fastest		Avg. time (sec.)	
	A/O	W-A/O	A/O	W-A/O
UNSAT (59)	19	**59**	110.56	**102.96**
SAT (36)	10	**29**	693.16	**569.73**

search completed fastest *tie* with Witness-AND/OR tree search. Indeed, Witness-AND/OR tree search was fastest on all 59 unsatisfiable instances. Looking at the average CPU time, Witness-AND/OR outperformed AND/OR tree search on both satisfiable and unsatisfiable instances.

Space requirements: Table C.6 gives the average number of stored goods and no-goods by AND/OR and Witness-AND/OR tree search. As discussed for the case of BTD, there are roughly the same number of no-goods stored for Witness-AND/OR and AND/OR tree search. However, the average number of goods stored is significantly reduced on both satisfiable and unsatisfiable instances. Because witness-based search dramatically reduces the space needed for caching, it directly benefits adaptive caching schemes to maintain more information cached than it would otherwise be possible [Marinescu and Dechter, 2006].

Table C.6: Average number of goods and no-goods stored

	A/O	W-A/O
Average #no-goods		
UNSAT(59)	9,645.76	**9,645.66**
SAT(36)	725,561.92	**711,190.75**
Average #goods		
UNSAT(59)	8,103.34	**5,783.95**
SAT(36)	103,506.00	**47,470.53**

In summary, Witness-AND/OR tree search is a beneficial strategy to implement and use in all circumstances and clearly improves AND/OR tree search.

C.5.4 An example with extreme benefits

While the average values of the results reported above show a clear advantage of the witness-based search, we explore below a situation where an extreme saving can be obtained.

Inspired by the experiments reported by Otten and Dechter [2012], we manually create an instance of a CSP that has a very large search space but is unsolvable. We show how Witness-AND/OR search can yield extreme gains. In practice, we proceed as follows. We connect a large search space many solutions to another search space with no solutions as illustrated in Figure C.9. To this end, we generate the pseudo-tree of each problem independently. We identify the root node of the barren search space. In the pseudo-tree of the solvable instance, we identify a variable that appears at the 'middle height' of the tree. Then, we add an arbitrary binary constraint between the two identified variables, thus linking the two CSP instances. We solve the newly formed instance with both AND/OR and Witness-AND/OR.

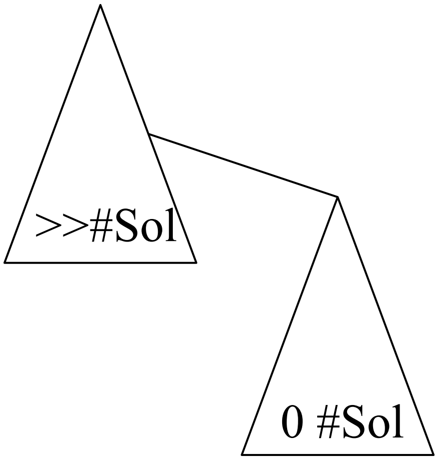


Figure C.9: Connecting a tree with no solution to a tree with many solutions

We generated one such problem by connecting an unsatisfiable aim-50 instance (normalized-aim50-1-6-unsat1.xml) to a pseudo-garden instance (normalized-g-9x9.xml) by adding an equality constraint between two variables (V53 of pseudo-garden to V1 of aim-50). The results were as follows:

1. AND/OR search expanded 2,657,758 nodes and detected unsolvability in 31.61

seconds.

2. Witness-AND/OR search reduces the effort by over 90%, visiting 63,476 nodes for a total of 2.25 seconds CPU time.

This example illustrates the significant advantage witness-based techniques can provide. Again, as stated earlier, this advantage does not cause any overhead.

Summary

In this chapter, we proposed witness-based search as a strategy to improve the time and space performance of solution-counting methods that operate on a tree structure. We empirically showed that our technique benefit solution-counting methods based on the BTD and AND/OR tree search improving performance by all measurements, especially the space needed for caching, which is a major bottleneck in such methods. As future work, we plan to extend our approach to approximate solution counting [Gogate and Dechter, 2008]. We believe that the space savings obtained by our witness strategy will allow us to achieve better approximations.

Appendix D

Assigning Blame when Triggering HLC

When search backtracks, the changes that the consistency algorithm made on the problem has to be undone. When a consistency algorithm is enforced uniformly at ever level of search, we can use the level of search to determine when that change was made. However, when a consistency algorithm is enforced selectively, the level of search does not accurately determine when the change was made.

We first present an example illustrating such a situation. Then, we propose two strategies for how to correctly assign blame. The first being an exact strategy, which will precisely determine where the blame occurs, and the second an approximation.

D.1 A Simple Motivating Example

We present motivating example to illustrate the situation where selectively applying higher-level consistency can lead to repeated work being done.

Example 7 *Consider that a high-level consistency (HLC) algorithm was ran at pre-*

processing. Search chooses to instantiate $A \leftarrow 1$*, but HLC was not enforced at this step. Search can then instantiate variables* $B \leftarrow 1$ *and* $C \leftarrow 1$ *through a domino effect (i.e.,* B *and* C *only had one value in their domain). At this point HLC is enforced for a second time and determines that* D *cannot take value* 1*. However, because* B *and* C *were instantiated using a domino effect the removal of* 1 *from* D *is attributed to the instantiation of* A *for backtracking purposes.*

Assigning the blame to the correct level is important because upon backtracking, all of the values removed by consistency are restored. In the situation of selectively running a consistency algorithm, determining the appropriate level of where to assign the blame, which may appear at a shallower depth in the search tree, saves on repeated work.

D.2 Apply Consistency at Each Step

The simplest approach to determining the correct depth of the search tree when a higher-level consistency could remove a value is to run the higher-level consistency at every depth of the search tree. In the case of selectively enforcing an HLC, restore the CSP to the state in which the HLC was last enforced. Re-apply the conditioning steps of search while enforcing the HLC algorithm at every step.

Such an approach has the advantage that the removal of values is attributed to the depth of search where the HLC first removed the value, but at the cost of many calls to the HLC algorithm.

D.3 An Approximation of Blame

We propose to use a single application of HLC and a heuristic to determine the level of search that could have caused the change, which we refer to as the 'blame.' The heuristic remains correct, meaning that it does not attribute the blame to a level prior to when it could be determined.

Our heuristic for assigning blame for a given change in the problem is by tracking the deepest level in search that caused every reduction. We break our discussion first on variable-based consistencies and then relational-based consistencies.

D.3.1 Variable-Based Consistencies

We address variable-based consistencies in the following manner. Each future variable is assigned a 'blame' variable that specifies the deepest variable that last modified the variable. Initially no variables are assigned, so the blame of every variable is none as no variable. Assigned variables are given a blame value of themselves. When a value is removed from a variable, the set of constraints that caused the removal are considered. The union of all of their scopes are considered, and the deepest blame variable from the union of scopes is then assigned to this removal. The blame variable of the variable is updated to the found blame variable.

The blame variable is similar to Prosser's [1993] 'past-fc[$\cdot$]' data-structure in Forward Checking and Conflict-Directed Backjumping (FC-CBJ), which stores a set of assigned variables that are responsible for the modification of a future variable. In our situation, we are storing the blame of variable x deepest variable of past-fc[x].

D.3.2 Relational-Based Consistencies

We address relation-based consistencies in the following manner. Each relation is assigned a ‘blame’ variable. Initially no variables are assigned and no values have been removed, thus the blame of every relation is none. When a tuple is removed from a relation, the set of constraints that caused the removal are considered. The union of all of their scopes are considered, and the deepest blame variable from the union of scopes is recorded as the blame variable for the relation.

D.3.3 Considering Both Relational and Variable-Based Consistencies

We address consistencies that are both relational and variable-based in the following manner. When considering the set of constraints that caused a removal, the deepest blame variable *both* the relations and the union of variables in the scopes should be considered.

It can easily be noticed that these techniques are only approximative of where the blame is to be assigned. The techniques over-approximate where the blame is to be assigned, and in certain situations could be placed earlier in the tree. However, they do not under-approximate, so the blame should not have appeared later. Further, the order of values removed can affect the location of the approximation.

Theorem 17 *The approximation when using variable-based consistencies is correct.*

Proof: (By contradiction) Assume that the approximation is not correct. That is, at search level i the consistency algorithm determines that the removal of $X \leftarrow v$ through the consideration of the constraints $C = \{c_1, c_2, \ldots, c_m\}$ is attributed to level $j < i$, but the real level is $j < k < i$.

Consider $S = \cup_{c_l \in C} scp(c_l)$. The blame of each of these variables in S at search level i will be $\leq j$. The variables were not modified between levels j and i. Thus, the information at level j could find the change, which contradicts that the level should be k. □

Theorem 18 *The approximation when using relational-based consistencies is correct.*

Proof: Follows by the same argument as Theorem 17. □

Theorem 19 *The approximation when using both relational and variable-based consistencies is correct.*

Proof: Follows by the same argument as Theorem 17. □

Notice, for POAC all of the variables are considered at any given time, thus, the blame will always be the current level.

Summary

In this chapter we introduced a strategy for determining the depth of search that filtering can be attributed to when using a triggering strategy. In particular, we identified an exact strategy and an approximation.

Appendix E

Benchmark Information

In this appendix, we provide information about the benchmarks from Lecoutre's website[1] used in this thesis. In particular, we report the primal graph density of the benchmarks and then report the performance of searching using GAC2001 [Bessière *et al.*, 2005] or STR2+ [Lecoutre, 2011] as RFL.

E.1 Primal Density of Benchmarks

Table E.1 shows the average, min, and max primal density for each benchmark. If the average density is greater less than 50%, the density is shown in gray to indicate that it is included in the experiments.

Table E.1: Primal densities for benchmark instances

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
Summary	5,998	9,549	5,484	0.1%	100.0%	36.8%
aim-100	3	24	0	7.2%	24.9%	12.4%

---

[1] `www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

. . . continued

Benchmark	Max Arity	# Instances	# Memout	Primal Density		
				Min	Max	Average
aim-200	3	24	0	3.7%	15.3%	6.8%
aim-50	3	24	0	14.0%	45.4%	23.2%
allIntervalSeries	3	25	2	50.9%	66.7%	55.6%
allsquares	38	37	37	-	-	-
allsquaresUnsat	38	37	37	-	-	-
bddLarge	15	35	0	100.0%	100.0%	100.0%
bddSmall	18	35	0	100.0%	100.0%	100.0%
BH-4-13	2	7	0	19.8%	19.8%	19.8%
BH-4-4	2	10	0	22.1%	22.1%	22.1%
BH-4-7	2	20	0	20.7%	20.7%	20.7%
bibd10-11	12	6	6	-	-	-
bibd12-13	132	7	7	-	-	-
bibd6	100	10	10	-	-	-
bibd7	98	14	14	-	-	-
bibd8	98	7	7	-	-	-
bibd9	12	10	10	-	-	-
bibdVariousK	21	29	29	-	-	-
bmc/	53	24	24	-	-	-
bqwh-15-106	2	100	0	11.6%	11.6%	11.6%
bqwh-18-141	2	100	0	9.8%	9.8%	9.8%
cabinet	14	40	40	-	-	-
chessboardColoration	4	20	2	100.0%	100.0%	100.0%
cjss	3	10	10	-	-	-
classes	9	100	100	-	-	-
coloring	2	22	0	5.7%	53.3%	19.2%
compet02	2	20	20	-	-	-
compet08	24	16	16	-	-	-
composed-25-1-2	2	10	0	42.4%	42.4%	42.4%
composed-25-1-25	2	10	0	46.8%	46.8%	46.8%
composed-25-1-40	2	10	0	49.6%	49.6%	49.6%
composed-25-1-80	2	10	0	57.2%	57.2%	57.2%

. . . continued

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
composed-25-10-20	2	10	0	11.4%	11.4%	11.4%
composed-75-1-2	2	10	0	18.3%	18.3%	18.3%
composed-75-1-25	2	10	0	19.0%	19.0%	19.0%
composed-75-1-40	2	10	0	19.5%	19.5%	19.5%
composed-75-1-80	2	10	0	20.6%	20.6%	20.6%
costasArray	2	11	11	-	-	-
cril	11	8	0	1.7%	100.0%	37.2%
domino	2	24	10	0.1%	2.0%	0.7%
driver	2	7	0	8.3%	11.2%	9.9%
dubois	3	13	0	1.3%	6.7%	4.9%
ehi-85	2	100	0	9.3%	9.4%	9.3%
ehi-90	2	100	0	8.8%	8.9%	8.9%
fapp01-05/fapp01	2	11	11	-	-	-
fapp01-05/fapp02	2	11	11	-	-	-
fapp01-05/fapp03	2	11	11	-	-	-
fapp01-05/fapp04	2	11	11	-	-	-
fapp01-05/fapp05	2	11	11	-	-	-
fapp06-10/fapp06	2	11	11	-	-	-
fapp06-10/fapp07	2	11	11	-	-	-
fapp06-10/fapp08	2	11	11	-	-	-
fapp06-10/fapp09	2	11	11	-	-	-
fapp06-10/fapp10	2	11	11	-	-	-
fapp11-15/fapp11	2	11	11	-	-	-
fapp11-15/fapp12	2	11	11	-	-	-
fapp11-15/fapp13	2	11	11	-	-	-
fapp11-15/fapp14	2	11	11	-	-	-
fapp11-15/fapp15	2	11	11	-	-	-
fapp16-20/fapp16	2	11	11	-	-	-
fapp16-20/fapp17	2	11	11	-	-	-
fapp16-20/fapp18	2	11	11	-	-	-
fapp16-20/fapp19	2	11	11	-	-	-

. . . continued

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
fapp16-20/fapp20	2	11	11	-	-	-
fapp21-25/fapp21	2	11	11	-	-	-
fapp21-25/fapp22	2	11	11	-	-	-
fapp21-25/fapp23	2	11	11	-	-	-
fapp21-25/fapp24	2	11	11	-	-	-
fapp21-25/fapp25	2	11	11	-	-	-
fapp26-30/fapp26	2	11	11	-	-	-
fapp26-30/fapp27	2	11	11	-	-	-
fapp26-30/fapp28	2	11	11	-	-	-
fapp26-30/fapp29	2	11	11	-	-	-
fapp26-30/fapp30	2	11	11	-	-	-
fapp31-35/fapp31	2	11	11	-	-	-
fapp31-35/fapp32	2	11	11	-	-	-
fapp31-35/fapp33	2	11	11	-	-	-
fapp31-35/fapp34	2	11	11	-	-	-
fapp31-35/fapp35	2	11	11	-	-	-
fapp36-40/fapp36	2	11	11	-	-	-
fapp36-40/fapp37	2	11	11	-	-	-
fapp36-40/fapp38	2	11	11	-	-	-
fapp36-40/fapp39	2	11	11	-	-	-
fapp36-40/fapp40	2	11	11	-	-	-
fischer	3	121	121	-	-	-
frb30-15	2	10	0	47.8%	49.9%	48.7%
frb35-17	2	10	0	43.7%	45.9%	44.5%
frb40-19	2	10	0	39.5%	41.8%	41.1%
frb45-21	2	10	0	37.3%	39.8%	38.3%
frb50-23	2	10	0	34.9%	37.2%	35.8%
frb53-24	2	10	0	34.1%	34.5%	34.4%
frb56-25	2	10	0	33.2%	34.1%	33.5%
frb59-26	2	10	0	31.5%	32.7%	32.1%
geom	2	100	0	27.7%	45.3%	34.4%

. . . continued

Benchmark	Max Arity	# Instances	# Memout	Primal Density		
				Min	Max	Average
golombRulerArity3	3	14	0	71.4%	79.0%	75.1%
golombRulerArity4	4	14	12	100.0%	100.0%	100.0%
graphs-valiente	64	793	775	100.0%	100.0%	100.0%
half	7	25	0	97.0%	99.7%	98.3%
hanoi	2	5	0	1.6%	33.3%	11.8%
haystacks	2	51	17	2.6%	22.5%	6.5%
hos/	2	14	5	0.7%	2.8%	1.7%
insertion/full-insertion	2	41	4	1.6%	23.0%	7.0%
insertion/k-insertion	2	32	0	1.0%	10.8%	4.3%
jnhSat	14	16	0	84.0%	89.9%	85.5%
jnhUnsat	11	34	0	83.9%	88.4%	86.3%
jobShop-e0ddr1	2	10	0	21.6%	21.6%	21.6%
jobShop-e0ddr2	2	10	0	21.6%	21.6%	21.6%
jobShop-enddr1	2	10	0	21.6%	21.6%	21.6%
jobShop-enddr2	2	6	0	21.6%	21.6%	21.6%
jobShop-ewddr2	2	10	0	21.6%	21.6%	21.6%
knights	2	19	19	-	-	-
langford	2	4	0	100.0%	100.0%	100.0%
langford2	2	24	1	100.0%	100.0%	100.0%
langford3	2	24	2	100.0%	100.0%	100.0%
langford4	2	24	4	100.0%	100.0%	100.0%
lard	2	10	0	100.0%	100.0%	100.0%
largeQueens	2	5	5	-	-	-
latinSquare	12	10	10	-	-	-
leighton/leighton-15	2	28	0	8.1%	16.6%	12.3%
leighton/leighton-25	2	32	0	8.2%	17.2%	12.7%
leighton/leighton-5	2	8	0	5.7%	9.7%	7.7%
lexHerald	34	47	47	-	-	-
lexPuzzle	2	22	21	100.0%	100.0%	100.0%
lexVg	20	63	0	10.7%	40.0%	19.4%
magicSquare	4	18	18	-	-	-

. . . continued

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
marc	2	11	0	100.0%	100.0%	100.0%
medium	2	5	5	-	-	-
mknap	15	6	4	100.0%	100.0%	100.0%
modifiedRenault	10	50	0	8.7%	9.6%	9.0%
mug/	2	8	0	3.4%	3.8%	3.6%
myciel	2	16	0	13.0%	36.4%	20.4%
nengfa	10	10	5	1.4%	100.0%	39.2%
ogdHerald	46	50	50	-	-	-
ogdPuzzle	2	22	21	100.0%	100.0%	100.0%
ogdVg	20	65	0	10.7%	40.0%	19.1%
ortholatin	4	9	9	-	-	-
os-gp-sat	2	10	10	-	-	-
os-gp-unsat	2	10	10	-	-	-
pigeons_glb	5	19	19	-	-	-
pigeons	2	25	0	100.0%	100.0%	100.0%
pret	3	8	0	2.7%	6.8%	4.7%
primes-10	5	32	30	9.1%	38.1%	23.6%
primes-15	3	32	31	38.1%	38.1%	38.1%
primes-20	3	32	31	38.1%	38.1%	38.1%
primes-25	3	32	31	38.1%	38.1%	38.1%
primes-30	3	32	31	38.1%	38.1%	38.1%
pseudo/aim	3	48	0	1.4%	16.9%	5.8%
pseudo/chnl	20	21	16	11.0%	17.4%	14.0%
pseudo/circuits	20	7	4	19.6%	52.2%	40.6%
pseudo/course	22	4	4	-	-	-
pseudo/fpga	20	36	15	7.5%	15.4%	11.5%
pseudo/garden	5	7	1	4.8%	100.0%	35.7%
pseudo/ii	10	41	27	0.5%	5.9%	1.5%
pseudo/jnh	14	16	0	37.2%	43.8%	38.7%
pseudo/logic-synthesis	18	17	16	1.6%	1.6%	1.6%
pseudo/mps	19	49	43	3.7%	100.0%	60.8%

. . . continued

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
pseudo/mpsReduced	5,998	106	106	-	-	-
pseudo/niklas	3,861	19	19	-	-	-
pseudo/par	3	30	30	-	-	-
pseudo/ppp	29	6	6	-	-	-
pseudo/primesDimacs	18	11	11	-	-	-
pseudo/radar	27	12	12	-	-	-
pseudo/routing	35	15	15	-	-	-
pseudo/ssa	5	8	8	-	-	-
pseudo/ttp	36	8	8	-	-	-
pseudo/uclid	25	39	39	-	-	-
pseudoGLB	7	384	384	-	-	-
QCP-10	2	15	0	18.2%	18.2%	18.2%
QCP-15	2	15	0	12.5%	12.5%	12.5%
QCP-20	2	15	0	9.5%	9.5%	9.5%
QCP-25	2	15	0	7.7%	7.7%	7.7%
QG3/	18	7	6	57.4%	57.4%	57.4%
QG4/	18	7	6	57.4%	57.4%	57.4%
QG5/	9	7	3	13.6%	25.4%	18.7%
QG6/	18	7	6	48.8%	48.8%	48.8%
QG7/	18	7	6	48.8%	48.8%	48.8%
queenAttacking	2	10	3	88.9%	98.2%	95.2%
queens	2	14	5	100.0%	100.0%	100.0%
queensKnights	2	18	5	100.0%	100.0%	100.0%
QWH-10	2	10	0	18.2%	18.2%	18.2%
QWH-15	2	10	0	12.5%	12.5%	12.5%
QWH-20	2	10	0	9.5%	9.5%	9.5%
QWH-25	2	10	0	7.7%	7.7%	7.7%
radar-8-24-3-2	19	50	50	-	-	-
radar-8-30-3-0	22	50	50	-	-	-
radar-9-28-4-2	24	50	50	-	-	-
ramsey3	3	8	0	15.4%	36.4%	23.9%

. . . continued

Benchmark	Max Arity	# Instances	# Memout	Primal Density		
				Min	Max	Average
ramsey4	3	8	0	8.7%	15.4%	11.7%
rand-10-20-10	10	20	0	73.7%	88.2%	79.1%
rand-10-60-20	10	50	0	52.0%	58.3%	54.4%
rand-2-23	2	10	0	100.0%	100.0%	100.0%
rand-2-24	2	10	0	100.0%	100.0%	100.0%
rand-2-25	2	10	0	100.0%	100.0%	100.0%
rand-2-26	2	10	0	100.0%	100.0%	100.0%
rand-2-27	2	10	0	100.0%	100.0%	100.0%
rand-2-30-15-fcd	2	50	0	47.8%	52.9%	51.1%
rand-2-30-15	2	50	0	47.8%	52.9%	51.1%
rand-2-40-19-fcd	2	50	0	41.7%	45.0%	43.4%
rand-2-40-19	2	50	0	41.7%	45.0%	43.4%
rand-2-50-23-fcd	2	50	0	37.1%	39.6%	38.1%
rand-2-50-23	2	50	0	37.1%	39.6%	38.1%
rand-3-20-20-fcd	3	50	0	55.8%	67.4%	61.3%
rand-3-20-20	3	50	0	55.8%	67.4%	61.3%
rand-3-24-24-fcd	3	50	0	52.2%	60.5%	56.4%
rand-3-24-24	3	50	0	52.2%	60.5%	56.4%
rand-3-28-28-fcd	3	50	0	48.9%	56.9%	52.6%
rand-3-28-28	3	50	0	48.9%	56.9%	52.6%
rand-5-12-12	5	50	0	100.0%	100.0%	100.0%
rand-8-20-5	8	20	0	91.1%	96.8%	94.8%
rand	15	25	0	99.6%	100.0%	100.0%
rcpsp	25	39	39	-	-	-
rcpspTighter	25	39	39	-	-	-
register/fpsol	2	37	7	13.2%	32.3%	19.6%
register/inithx	2	32	10	9.0%	13.9%	9.9%
register/mulsol	2	49	0	25.8%	41.5%	30.1%
register/zeroin	2	31	0	28.9%	52.1%	38.6%
renault	10	2	0	10.0%	10.0%	10.0%
rlfapGraphs	2	14	0	1.1%	5.7%	2.7%

. . . continued

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
rlfapGraphsMod	2	12	0	0.4%	100.0%	17.9%
rlfapScens11	2	12	0	1.8%	1.8%	1.8%
rlfapScens	2	11	0	1.7%	6.6%	3.1%
rlfapScensMod	2	13	0	0.9%	6.2%	3.0%
school	2	8	0	24.8%	26.9%	25.8%
schurrLemma	3	10	0	97.0%	99.7%	98.7%
sgb/book	2	26	0	1.1%	12.7%	7.4%
sgb/games	2	4	0	8.9%	8.9%	8.9%
sgb/miles	2	42	0	7.8%	64.0%	37.8%
sgb/queen	2	50	0	19.4%	53.3%	28.1%
si2-bvg	2	360	360	-	-	-
si2-m4D	259	120	120	-	-	-
si2-rand	12	120	120	-	-	-
si4-bvg	4	360	360	-	-	-
si4-m4D	518	120	120	-	-	-
si4-rand	24	120	120	-	-	-
si6-bvg	6	360	360	-	-	-
si6-m4D	777	120	120	-	-	-
si6-rand	36	120	120	-	-	-
small	19	5	5	-	-	-
socialGolfers	24	12	6	1.1%	2.0%	1.5%
ssa/	6	8	1	0.3%	16.2%	3.8%
subs	2	9	0	45.0%	83.1%	67.2%
super-jobShop-e0ddr1	2	10	0	17.1%	17.1%	17.1%
super-jobShop-e0ddr2	2	10	0	17.1%	17.1%	17.1%
super-jobShop-enddr1	2	10	0	17.1%	17.1%	17.1%
super-jobShop-enddr2	2	6	0	17.1%	17.1%	17.1%
super-jobShop-ewddr2	2	10	0	17.1%	17.1%	17.1%
super-js-taillard-15	2	30	30	-	-	-
super-js-taillard-20-15	2	30	30	-	-	-
super-js-taillard-20	2	30	30	-	-	-

. . . continued

Benchmark	**Max Arity**	**# Instances**	**# Memout**	**Primal Density**		
				Min	**Max**	**Average**
super-os-taillard-10	2	30	30	-	-	-
super-os-taillard-15	2	30	30	-	-	-
super-os-taillard-20	2	30	30	-	-	-
super-os-taillard-4	2	30	0	32.3%	32.3%	32.3%
super-os-taillard-5	2	30	0	26.5%	26.5%	26.5%
super-os-taillard-7	2	30	30	-	-	-
super-queens	2	14	6	75.3%	78.6%	76.2%
tdsp	6	42	42	-	-	-
test01-04/test01	2	11	11	-	-	-
test01-04/test02	2	11	11	-	-	-
test01-04/test03	2	11	11	-	-	-
test01-04/test04	2	11	11	-	-	-
tightness0.1	2	100	0	96.5%	96.5%	96.5%
tightness0.2	2	100	0	53.1%	53.1%	53.1%
tightness0.35	2	100	0	32.1%	32.1%	32.1%
tightness0.5	2	100	0	23.1%	23.1%	23.1%
tightness0.65	2	100	0	17.3%	17.3%	17.3%
tightness0.8	2	100	0	13.2%	14.5%	13.2%
tightness0.9	2	100	0	10.8%	11.8%	10.8%
travellingSalesman-20	3	15	0	14.8%	14.8%	14.8%
travellingSalesman-25	3	15	0	14.0%	14.0%	14.0%
ukHerald	38	50	50	-	-	-
ukPuzzle	2	22	21	100.0%	100.0%	100.0%
ukVg	20	65	0	10.7%	40.0%	19.1%
varDimacs	10	9	0	0.5%	28.6%	15.5%
wordsHerald	38	49	49	-	-	-
wordsPuzzle	2	22	21	100.0%	100.0%	100.0%
wordsVg	20	65	0	10.7%	40.0%	19.1%

E.2 Performance of GAC2001 and STR2+ on Binary CSPs

Table E.2 report the performance of searching using GAC2001 [Bessière *et al.*, 2005] or STR2+ [Lecoutre, 2011] as RFL.

Table E.2: Performance of GAC2001 and STR2+ on Binary CSPs

Benchmark	**#Instances**	**#Solved**		**∑CPU**		**#NV**
		GAC2001	**STR2+**	**GAC2001**	**STR2+**	
Summary	2,125	1,750	1,725	114,565.6	>303,622.2	108,758.7
BH-4-4	10	10	10	419.6	723.6	428,832.0
QCP-10	15	15	15	3.6	4.7	520.2
QCP-15	15	15	15	668.7	1,310.0	235,194.5
QCP-20	15	5	4	2,103.3	>5,328.7	770,529.5
QCP-25	15	1	1	21.9	25.8	4,295.0
QWH-10	10	10	10	1.8	2.1	327.6
QWH-15	10	10	10	22.9	34.4	6,814.1
QWH-20	10	9	9	1,292.5	2,581.3	532,566.1
QWH-25	10	0	0	-	-	-
bqwh-15-106	100	100	100	32.9	56.6	3,185.0
bqwh-18-141	100	100	100	425.4	814.0	42,308.0
coloring	22	22	22	397.4	734.1	343,874.5
composed-25-1-2	10	10	10	1.0	1.6	517.9
composed-25-1-25	10	10	10	1.1	1.8	446.1
composed-25-1-40	10	10	10	1.2	2.0	418.4
composed-25-1-80	10	10	10	1.3	2.2	258.8
composed-25-10-20	10	10	10	2.2	3.2	663.8
composed-75-1-2	10	10	10	2.7	4.6	1,106.4
composed-75-1-25	10	10	10	3.0	5.1	1,026.1
composed-75-1-40	10	10	10	3.1	5.3	869.3
composed-75-1-80	10	10	10	3.1	4.8	511.8
domino	24	14	12	2,823.8	>7,916.3	600.0

. . . continued

Benchmark	**#Instances**	**#Solved**		**∑CPU**		**#NV**
		GAC2001	**STR2+**	**GAC2001**	**STR2+**	
driver	7	7	7	55.6	83.8	6,825.1
ehi-85	100	100	100	250.6	360.6	1,397.9
ehi-90	100	100	100	263.2	369.8	1,235.4
frb30-15	10	10	10	15.0	36.5	3,607.8
frb35-17	10	10	10	149.9	383.9	32,255.6
frb40-19	10	10	10	1,295.4	3,347.9	253,514.0
frb45-21	10	8	7	6,151.3	>17,642.0	1,268,086.4
frb50-23	10	2	2	625.7	1,622.8	477,914.0
geom	100	100	100	2,439.1	7,254.3	27,235.8
hanoi	5	5	5	30.6	2.4	47.6
hos	14	12	11	1,060.0	>4,728.6	3,830.6
insertion/full-insertion	41	32	32	1,979.3	3,923.3	65,648.6
insertion/k-insertion	32	16	16	206.1	355.2	367,761.6
jobShop-e0ddr1	10	5	5	363.3	2,683.0	57,132.0
jobShop-e0ddr2	10	6	4	2,480.7	>7,244.6	50.0
jobShop-enddr1	10	9	9	150.4	196.9	2,009.3
jobShop-enddr2	6	4	3	1,601.4	>3,637.0	50.0
jobShop-ewddr2	10	10	10	235.6	129.9	50.0
knights	19	0	0	-	-	-
leighton/leighton-25	32	6	6	95.8	109.1	286.7
leighton/leighton-5	8	8	8	45.1	54.2	495.5
mug	8	4	4	.1	.1	94.0
myciel	16	13	13	1,817.2	3,161.1	985,349.7
rand-2-30-15-fcd	50	50	50	96.8	232.2	5,054.8
rand-2-30-15	50	50	50	169.8	415.9	8,999.1
rand-2-40-19-fcd	50	50	49	10,098.7	>25,093.2	332,287.0
rand-2-40-19	50	50	49	20,323.0	>51,055.4	723,885.4
rand-2-50-23-fcd	50	7	2	9,922.5	>23,212.3	1,359,944.0
rand-2-50-23	50	4	0	6,827.1	>14,400.0	-
register/fpsol	37	5	5	79.0	84.9	298.2
register/inithx	32	5	5	160.6	174.7	376.2

. . . continued

Benchmark	**#Instances**	**#Solved**		**∑CPU**		**#NV**
		GAC2001	**STR2+**	**GAC2001**	**STR2+**	
register/mulsol	49	10	10	83.9	88.3	294.9
rlfapGraphs	14	14	14	154.2	136.4	344.4
rlfapGraphsMod	12	12	12	138.9	394.3	12,350.0
rlfapScens11	12	6	5	2,282.2	>6,076.0	59,473.6
rlfapScens	11	11	11	140.3	130.4	514.5
rlfapScensMod	13	13	13	80.7	165.1	2,637.2
school	8	3	3	80.3	85.7	363.7
sgb/book	26	23	22	5,052.1	>10,337.2	1,367,470.9
sgb/games	4	4	4	230.0	600.8	882,427.3
sgb/queen	50	15	14	3,146.9	>9,312.9	189,142.9
super-os/super-os-taillard-4	30	28	28	1,951.1	7,647.2	6,713.1
super-os/super-os-taillard-5	30	12	9	5,359.5	>22,771.1	7,153.8
tightness0.35	100	100	100	3,067.4	7,547.1	80,978.6
tightness0.5	100	100	100	4,264.9	11,562.0	100,557.5
tightness0.65	100	100	100	3,402.3	10,056.7	62,131.6
tightness0.8	100	100	100	3,352.5	10,839.2	36,255.3
tightness0.9	100	100	100	4,557.2	14,314.1	25,658.8

Appendix F

Detailed Results for Chapter 4

Table F.1 shows detailed results for Chapter 4.

Table F.1: All benchmark data sorted by PrePeak$^+$ CPU time gain over STR

		# solved by					$\sum$CPU [sec]					# Calls POAC			
Benchmark	# Instances	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
pseudo-ii	41	9	14	14	13	12	>18,619.8	2,481.9	2,088.4	>5,273.4	>8,749.7	18,202.9	0.0	0.0	0.0
dubois	13	6	11	10	7	8	>22,289.1	2,550.8	>13,143.0	>17,855.7	>16,627.1	201,177.3	53,286.3	67,505.2	39,506.2
mug	8	4	6	5	5	6	>7,200.1	2,974.8	>4,291.2	>4,095.1	3,249.7	86.0	0.0	0.0	0.0
QCP-20	15	4	4	5	4	4	>5,328.7	>4,861.0	2,762.9	>5,090.7	>5,711.3	11,245.5	603.3	688.0	16.3
nengfa	10	4	4	5	5	5	>3,820.6	>4,235.9	2,321.0	2,331.4	2,612.5	1,959.0	0.0	0.0	0.0
frb45-21	10	7	0	8	7	7	>17,642.0	>28,800.0	16,239.8	>17,047.1	>17,055.2	-	-	-	-
k-insertion	32	16	17	17	17	16	>3,955.2	3,550.0	2,903.5	3,028.9	>3,932.3	7,883.3	1,745.8	2,050.3	729.9
pseudo-aim	48	48	48	48	48	48	868.1	222.2	296.0	406.7	448.3	1,954.5	224.8	410.8	343.8
QWH-20	10	9	9	9	9	9	2,581.3	1,870.9	2,102.9	2,097.4	2,349.2	8,049.3	145.7	7.6	4.8
pseudo-fpga	36	2	2	2	2	2	2,047.9	3,011.9	1,937.9	1,717.3	1,799.0	2,897,334.0	98,490.5	214.0	134.5
ssa	8	7	7	7	7	7	180.8	62.9	79.0	89.4	89.6	1,699.3	40.1	20.6	20.4
QCP-15	15	15	15	15	15	15	1,310.0	1,248.4	1,213.5	1,119.1	1,182.1	6,349.3	302.9	7.5	6.6
bqwh-18-141	100	100	100	100	100	100	814.0	772.5	772.2	899.2	815.1	955.4	50.1	15.7	26.3
sgb-queen	50	14	12	14	14	14	5,712.9	>9,969.6	5,692.0	5,728.2	5,703.2	15,033.3	9.3	3.7	3.8
aim-200	24	24	24	24	24	24	49.5	18.2	34.3	37.2	34.8	773.1	90.1	106.4	77.3
rand-3-24-24	50	5	0	5	5	5	11,650.2	>18,000.0	11,638.9	11,638.0	11,638.1	-	-	-	-
coloring	22	22	22	22	22	22	734.1	1,545.2	725.8	809.2	866.3	37,397.0	4.2	3.8	3.4
super-os-taillard-5	30	9	1	9	9	7	11,971.1	>28,924.3	11,969.8	11,983.0	>14,219.0	1.0	0.0	0.0	0.0
driver	7	7	7	7	7	7	83.8	248.8	83.5	90.0	89.7	52.0	0.0	0.0	0.0

. . . continued

Benchmark	# Instances	# solved by					∑CPU [sec]					# Calls POAC			
		GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
super-jobShop-e0ddr1	10	2	0	2	2	2	55.3	>7,200.0	55.2	55.4	55.4	-	-	-	-
pseudo-circuits	7	3	3	3	3	3	11.0	32.7	11.0	11.0	11.0	24.0	0.0	0.0	0.0
QWH-25	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
bqwh-15-106_glb	100	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
bqwh-18-141_glb	100	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
cjss	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
compet02	20	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
compet08	16	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
costasArray	11	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
fischer	121	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
frb53-24	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
frb56-25	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
frb59-26	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
haystacks	51	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
knights	19	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
largeQueens	5	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
latinSquare	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
magicSquare	18	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
medium	5	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
ogdHerald	50	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-

. . . continued

Benchmark	# Instances	# solved by					∑CPU [sec]					# Calls POAC			
		GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
ortholatin	9	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
os-gp-sat	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
os-gp-unsat	10	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	-	-	-	-
rcpsp	39	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
rcpspTighter	39	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
small	5	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
super-jobShop-e0ddr2	10	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
tdsp	42	0	0	0	0	0	.0	.0	.0	.0	.0	-	-	-	-
jobShop-enddr2	6	3	0	3	3	3	37.0	>10,800.0	37.0	37.1	37.2	-	-	-	-
primes-15	32	1	1	1	1	1	.0	.0	.0	.0	.0	140.0	0.0	0.0	0.0
primes-20	32	1	1	1	1	1	.0	.0	.0	.0	.0	139.0	0.0	0.0	0.0
primes-25	32	1	1	1	1	1	.0	.1	.0	.0	.0	141.0	0.0	0.0	0.0
primes-30	32	1	1	1	1	1	.0	.1	.0	.0	.0	140.0	0.0	0.0	0.0
renault	2	2	2	2	2	2	1.9	47.3	1.9	1.9	1.9	21.0	0.0	0.0	0.0
ramsey3	8	2	2	2	2	2	.1	.1	.1	.1	.1	48.5	0.0	0.0	0.0
primes-10	32	2	2	2	2	2	.6	5.1	.6	.6	.6	125.5	0.0	0.0	0.0
composed-25-1-80	10	10	10	10	10	10	2.2	2.5	2.2	2.2	2.2	1.0	0.0	0.0	0.0
jobShop-e0ddr2	10	4	0	4	4	4	44.6	>14,400.0	44.6	44.8	44.7	-	-	-	-
pseudo-garden	7	6	6	6	6	6	.1	.2	.1	.2	.2	42.3	0.0	0.0	0.0

. . . continued

Benchmark	# Instances	# solved by					∑CPU [sec]					# Calls POAC			
		GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
composed-25-1-2	10	10	10	10	10	10	1.6	5.6	1.7	1.7	1.7	1.0	0.0	0.0	0.0
composed-25-1-25	10	10	10	10	10	10	1.8	5.4	1.9	1.9	1.9	1.0	0.0	0.0	0.0
composed-25-1-40	10	10	10	10	10	10	2.0	4.5	2.1	2.1	2.1	1.0	0.0	0.0	0.0
ramsey4	8	1	1	1	1	1	1.0	2.1	1.1	1.3	1.3	251.0	0.0	0.0	0.0
hanoi	5	5	5	5	5	5	2.4	2.5	2.5	2.5	2.5	1.0	0.0	0.0	0.0
jobShop-enddr1	10	9	9	9	9	9	196.9	17,936.8	197.0	197.2	197.2	112.7	0.0	0.0	0.0
composed-75-1-80	10	10	10	10	10	10	4.8	4.5	4.9	5.1	5.1	1.0	0.0	0.0	0.0
composed-25-10-20	10	10	10	10	10	10	3.2	26.3	3.3	3.5	3.5	46.6	0.0	0.0	0.0
QWH-10	10	10	10	10	10	10	2.1	4.0	2.2	2.5	2.5	29.4	0.0	0.0	0.0
super-jobShop-ewddr2	10	6	0	6	6	6	260.1	>21,600.0	260.2	260.9	260.9	-	-	-	-
jobShop-ewddr2	10	10	0	10	10	10	129.9	>36,000.0	130.0	130.2	130.3	-	-	-	-
aim-50	24	24	24	24	24	24	.6	.7	.7	.8	.8	24.7	0.0	0.0	0.0
composed-75-1-40	10	10	10	10	10	10	5.3	13.1	5.4	5.6	5.6	1.0	0.0	0.0	0.0
pseudo-logic-synthesis	17	1	1	1	1	1	17.5	57.5	17.6	18.0	17.9	916.0	0.0	0.0	0.0
composed-75-1-25	10	10	10	10	10	10	5.1	15.8	5.3	5.4	5.4	1.0	0.0	0.0	0.0
composed-75-1-2	10	10	10	10	10	10	4.6	18.8	4.8	4.9	4.9	1.0	0.0	0.0	0.0
register-zeroin	31	6	5	6	6	6	56.4	>6,752.3	56.6	57.4	57.4	155.2	0.0	0.0	0.0
sgb/miles	42	10	8	10	10	10	130.2	>7,829.8	130.4	131.0	131.1	597.5	0.0	0.0	0.0
QCP-10	15	15	15	15	15	15	4.7	9.2	4.9	5.3	5.4	78.2	0.0	0.0	0.0
super-jobShop-enddr2	6	2	0	2	2	2	201.2	>7,200.0	201.4	201.5	201.5	-	-	-	-

. . . continued

Benchmark	# Instances	# solved by					$\sum$CPU [sec]					# Calls POAC			
		GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
QCP-25	15	1	1	1	1	1	25.8	54.9	26.0	28.4	28.4	1,587.0	0.0	0.0	0.0
pseudo-mps	49	7	7	7	7	7	2,412.2	2,485.9	2,412.5	2,414.1	2,414.1	159.7	0.0	0.0	0.0
rlfapScens	11	11	10	11	11	11	130.4	>7,693.9	130.8	136.3	136.4	124.7	0.0	0.0	0.0
rlfapScensMod	13	13	13	13	13	13	165.1	2,344.7	165.5	168.2	168.2	338.5	0.0	0.0	0.0
modifiedRenault	50	50	50	50	50	50	70.9	456.1	71.3	71.5	71.5	21.9	0.0	0.0	0.0
aim-100	24	24	24	24	24	24	4.0	3.5	4.5	4.7	4.7	206.4	2.8	1.1	1.0
register-mulsol	49	10	8	10	10	10	88.3	>12,275.8	88.8	90.9	90.9	251.3	0.0	0.0	0.0
school	8	3	3	3	3	3	85.7	4,654.0	86.2	90.8	91.1	95.7	0.0	0.0	0.0
rlfapGraphs	14	14	11	14	14	14	136.4	>16,309.8	136.9	144.6	144.4	108.8	0.0	0.0	0.0
leighton-leighton-25	32	6	6	6	6	6	109.1	3,602.4	109.7	117.7	117.9	202.3	0.0	0.0	0.0
register-fpsol	37	5	3	5	5	5	84.9	>7,430.6	85.5	89.9	89.7	86.0	0.0	0.0	0.0
QWH-15	10	10	10	10	10	10	34.4	39.5	35.0	37.0	37.0	248.7	0.0	0.0	0.0
leighton-leighton-5	8	8	8	8	8	8	54.2	171.1	54.8	62.2	62.1	51.0	0.0	0.0	0.0
bqwh-15-106	100	100	100	100	100	100	56.6	80.6	57.4	59.6	59.5	145.9	0.1	0.1	0.1
domino	24	12	12	12	12	12	716.3	717.7	717.1	718.9	718.9	1.0	0.0	0.0	0.0
register-inithx	32	5	3	5	5	5	174.7	>7,663.6	175.5	186.2	186.3	86.0	0.0	0.0	0.0
pseudo-jnh	16	16	16	16	16	16	22.9	43.4	23.7	24.6	24.6	71.6	0.0	0.0	0.0
rlfapGraphsMod	12	12	12	12	12	12	394.3	2,044.6	395.8	402.7	402.8	1,082.4	0.0	0.0	0.0
rand-3-28-28-fcd	50	2	0	2	2	2	2,670.8	>7,200.0	2,672.7	2,671.7	2,671.7	-	-	-	-
leighton-leighton-15	28	6	6	6	6	6	695.1	2,717.2	697.5	705.1	707.3	7,404.2	0.0	0.0	0.0

. . . continued

Benchmark	# Instances	# solved by					$\sum$CPU [sec]					# Calls POAC			
		GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
frb30-15	10	10	10	10	10	10	36.5	162.5	39.5	37.4	37.2	364.4	3.2	1.6	1.3
rlfapScens11	12	5	3	5	5	5	2,476.0	>8,394.1	2,479.4	2,482.8	2,482.4	898.0	0.0	0.0	0.0
super-jobShop-enddr1	10	2	0	2	2	2	569.7	>7,200.0	573.6	570.1	570.2	-	-	-	-
rand-2-30-15	50	50	50	50	50	50	415.9	2,007.0	420.1	417.9	418.0	908.1	2.3	1.9	1.9
rand-2-30-15-fcd	50	50	50	50	50	50	232.2	1,080.0	236.8	234.1	233.9	525.5	1.7	1.4	1.2
rand-3-28-28	50	1	0	1	1	1	1,275.4	>3,600.0	1,280.1	1,276.4	1,289.8	-	-	-	-
ehi-85	100	100	100	100	100	100	360.6	218.2	365.7	399.2	399.2	1.0	0.0	0.0	0.0
ehi-90	100	100	100	100	100	100	369.8	242.0	374.9	412.2	412.1	1.0	0.0	0.0	0.0
socialGolfers	12	1	1	1	1	1	275.6	93.2	284.2	287.7	288.5	11,214.0	0.0	0.0	0.0
sgb-games	4	4	4	4	4	4	600.8	2,096.2	621.2	620.8	621.1	364,279.3	4.8	3.5	3.5
BH-4-4	10	10	10	10	10	10	723.6	2,365.1	744.7	742.7	743.8	79,646.0	21.0	15.0	15.0
travellingSalesman-20	15	15	15	15	15	15	276.5	1,426.9	298.6	286.0	271.3	963.2	10.3	11.8	8.2
super-os-taillard-4	30	28	22	28	28	28	7,647.2	>33,042.7	7,675.5	7,674.5	7,674.4	13.9	0.3	5.2	5.2
frb50-23	10	2	0	2	2	2	1,622.8	>7,200.0	1,653.3	1,643.2	1,654.4	-	-	-	-
frb35-17	10	10	10	10	10	10	383.9	1,846.7	414.8	394.9	398.4	3,074.8	25.2	12.0	12.2
cril	8	4	6	4	4	4	>12,655.6	>4,199.3	>12,706.3	>12,713.7	>12,702.2	73,552.3	52.3	59.7	37.7
full-insertion	41	32	30	32	32	32	3,923.3	>11,585.6	3,978.1	3,952.8	3,956.1	11,443.2	0.2	0.2	0.2
frb40-19	10	10	8	10	10	10	3,347.9	>12,051.4	3,443.0	3,396.8	3,388.1	8,541.3	47.6	23.9	19.6
ukVg	65	36	34	36	36	36	7,047.1	>18,255.8	7,142.1	7,067.8	7,079.0	153.8	0.6	0.9	0.4
sgb-book	26	22	20	22	22	22	6,737.2	>13,243.4	6,832.9	6,674.5	6,693.2	131,233.5	6.2	5.2	5.1

. . . continued

Benchmark	# Instances	# solved by					∑CPU [sec]					# Calls POAC			
		GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	GAC	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$	APOAC	PrePeak$^+$	BTWatch$^+$	PP-BTWatch$^+$
pret	8	4	4	4	4	4	387.3	152.9	485.8	486.7	493.5	218,033.5	108.8	74.3	84.0
myciel	16	13	12	13	12	13	3,161.1	>7,115.9	3,270.1	>5,212.7	3,214.9	90,943.1	11.4	12.8	9.3
ogdVg	65	41	35	41	40	41	12,075.4	>32,198.1	12,191.9	>13,136.9	12,839.8	57.6	0.3	0.2	0.2
geom	100	100	98	100	100	100	7,254.3	>28,365.6	7,372.8	7,318.0	7,316.1	2,694.0	5.4	2.9	2.9
tightness0.9	100	100	97	100	100	100	14,314.1	>58,017.9	14,446.9	14,442.9	14,460.2	222.2	8.5	7.5	7.2
rand-2-50-23-fcd	50	2	0	1	1	2	5,212.3	>7,200.0	>5,434.7	>5,408.8	5,254.6	-	-	-	-
lexVg	63	63	63	63	63	63	2,153.9	9,345.0	2,383.8	2,194.5	2,182.6	968.6	29.4	6.0	5.4
travellingSalesman-25	15	15	13	15	15	15	4,307.4	>11,339.7	4,590.6	5,270.8	4,511.1	3,068.2	51.8	39.8	20.1
tightness0.35	100	100	100	100	100	100	7,547.1	32,155.0	7,942.3	7,672.5	7,671.6	7,287.0	43.7	17.5	14.8
rand-2-40-19-fcd	50	49	40	49	49	49	21,493.2	>84,563.9	21,987.0	21,696.4	21,688.0	19,613.8	63.7	30.0	23.4
tightness0.8	100	100	99	100	100	100	10,839.2	>47,401.1	11,425.3	10,961.4	10,957.2	1,157.5	30.0	13.7	12.2
rand-3-24-24-fcd	50	14	4	14	15	14	>22,545.4	>44,910.1	>23,441.4	21,886.8	>23,302.7	5,061.3	5.0	5.0	5.0
tightness0.5	100	100	100	100	100	100	11,562.0	50,109.6	12,473.3	11,716.7	11,684.1	7,327.3	61.5	19.4	18.3
jobShop-e0ddr1	10	5	4	4	4	4	2,683.0	>10,728.2	>3,633.3	>3,633.4	>3,633.4	37.3	0.0	0.0	0.0
varDimacs	9	9	8	8	9	9	2,702.0	>3,931.3	>3,811.3	2,872.4	2,879.3	67,760.4	11.4	11.9	11.5
tightness0.65	100	100	100	100	100	100	10,056.7	40,771.5	11,455.9	10,167.7	10,199.7	4,151.2	64.6	19.4	15.3
rand-2-40-19	50	49	22	48	47	48	47,455.4	>134,187.4	>49,360.7	>50,492.6	>49,033.7	35,268.0	88.7	35.3	26.4
hos	14	11	10	10	10	10	1,128.6	>4,331.7	>3,973.0	>3,994.2	>3,994.1	576.1	0.0	0.0	0.0
wordsVg	65	65	61	64	65	65	8,399.7	>34,672.3	>15,901.7	8,501.6	8,502.2	1,478.3	168.3	5.3	4.0

Bibliography

[Amaldi *et al.*, 2010] Edoardo Amaldi, Claudio Iuliano, and Romeo Rizzi. Efficient Deterministic Algorithms for Finding a Minimum Cycle Basis in Undirected Graphs. In *Integer Programming and Combinatorial Optimization (IPCO 2010)*, volume 6080 of *LNCS*, pages 397–410, 2010.

[Balafrej *et al.*, 2013] Amine Balafrej, Christian Bessiere, Remi Coletta, and El-Houssine Bouyakhf. Adaptive Parameterized Consistency. In *Proceedings of* 19 th *International Conference on Principle and Practice of Constraint Programming (CP'13)*, volume 8124 of *LNCS*, pages 143–158. Springer, 2013.

[Balafrej *et al.*, 2014] Amine Balafrej, Christian Bessiere, El-Houssine Bouyakhf, and Gilles Trombettoni. Adaptive Singleton-Based Consistencies. In *Proceedings of AAAI-2014*, pages 2601–2607, Quebec City, Quebec, 2014.

[Balafrej *et al.*, 2015] Amine Balafrej, Christian Bessière, and Anastasia Paparrizou. Multi-Armed Bandits for Adaptive Constraint Propagation. In *Proceedings of the 24* th *International Joint Conference on Artificial Intelligence*, pages 290–296, Buenos Aires, Argentina, 2015.

[Baptiste *et al.*, 2006] Philippe Baptiste, Philippe Labori, Claude Le Pape, and Wim Nuijten. *Handbook of Constraint Programming*, chapter Constraint-Based Scheduling and Planning, pages 761–799. Elsevier, 2006.

[Bayardo and Mirankar, 1996] Roberto J. Bayardo and Daniel P. Mirankar. A Complexity Analysis of Space-Bound Learning Algorithms for the Constraint Satisfaction Problem. In *Proceedings of the Thirteen National Conference on Artificial Intelligence (AAAI 1996)*, pages 298–304, 1996.

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

[Bennaceur and Affane, 2001] Hachemi Bennaceur and Mohamed-Salah Affane. Partition-k-AC: An Efficient Filtering Technique Combining Domain Partition and Arc Consistency. In *Proceedings of* 7 *th International Conference on Principle and Practice of Constraint Programming (CP'01)*, volume 2239 of *LNCS*, pages 560–564. Springer, 2001.

[Bessière and Régin, 1996] Christian Bessière and Jean-Charles Régin. MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems. In *Proceedings of* 2 *nd International Conference on Principle and Practice of Constraint Programming (CP'96)*, volume 1118 of *LNCS*, pages 61–75. Springer, 1996.

[Bessière *et al.*, 2005] Christian Bessière, Jean-Charles Régin, Roland H.C. Yap, and Yuanlin Zhang. An Optimal Coarse-Grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

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

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

[Bitner and Reingold, 1975] James R. Bitner and Edward M. Reingold. Backtrack Programming Techniques. *Communications of the ACM*, 18(11):651–656, November 1975.

[Bliek and Sam-Haroud, 1999] 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.

[Borrett *et al.*, 1996] James E. Borrett, Edward P.K. Tsang, and Natasha R. Walsh. Adaptive Constraint Satisfaction: The Quickest First Principle. In *Proceedings of the 12 th European Conference on Artificial Intelligence*, pages 160–164, Budapest, Hungary, 1996.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting Systematic Search by Weighting Constraints. In *Proceedings of the 16 th European Conference on Artificial Intelligence*, pages 146–150, 2004.

[Carro and Hermenegildo, 1998] Manuel Carro and Manuel Hermenegildo. Some Design Issues in the Visualization of Constraint Logic Program Execution. In *In AGP'98 Joint Conference on Declarative Programming*, pages 71–86, 1998.

[Carro and Hermenegildo, 2000] Manuel Carro and Manuel Hermenegildo. Tools for Constraint Visualisation: The VIFID/TRIFID Tool. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, volume 1870 of *LNCS*, pages 253–272. Springer, 2000.

[Cohen and Jeavons, 2017] David A. Cohen and Peter G. Jeavons. The Power of Propagation: when GAC is Enough. *Constraints*, 22(1):3–23, Jan 2017.

[Darwiche, 2001] Adnan Darwiche. Recursive Conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.

[Davis and Putnam, 1960] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, July 1960.

[Debruyne and Bessière, 1997a] Romuald Debruyne and Christian Bessière. From Restricted Path Consistency to Max-Restricted Path Consistency. In *Proceedings of* 3 rd *International Conference on Principle and Practice of Constraint Programming (CP'97)*, volume 1330 of *LNCS*, pages 312–326. Springer, 1997.

[Debruyne and Bessière, 1997b] 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.

[Debruyne, 1999] Romuald Debruyne. A Strong Local Consistency for Constraint Satisfaction. In *Proceedings of the IEEE* 11 th *International Conference on Tools with Artificial Intelligence*, pages 202–209, 1999.

[Dechter and Mateescu, 2004] Rina Dechter and Robert Mateescu. The Impact of AND/OR Search Spaces on Constraint Satisfaction and Counting. In *Proceedings of* 10 th *International Conference on Principle and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 731–736. Springer, 2004.

[Dechter and Mateescu, 2006] Rina Dechter and Robert Mateescu. AND/OR Search Spaces for Graphical Models. *Artificial Intelligence*, 171(2-3):73–106, 2006.

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

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

[Dechter, 2003a] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

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

[Dechter, 2004] Rina Dechter. AND/OR Search Spaces for Graphical Models. Technical report, University of California, Irvine, 2004.

[Demeulenaere *et al.*, 2016] 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 *Proceedings of 22 nd International Conference on Principle and Practice of Constraint Programming (CP'16)*, volume 9892 of *LNCS*, pages 207–223. Springer, 2016.

[Eén and Biere, 2005] Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of Theory and Applications of Satisfiability Testing: 8th International Conference (SAT 2005)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.

[Epstein *et al.*, 2002] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. The Adaptive Constraint Engine. In *Proceedings of 8 th International Conference on Principle and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 525–540. Springer, 2002.

[Epstein *et al.*, 2005] Susan L. Epstein, Eugene C. Freuder, Richard M. Wallace, and Xingjian Li. Learning Propagation Policies. In *Second International Workshop on Constraint Propagation and Implementation, Volume I held in conjunction with CP 2005*, pages 1–15, 2005.

[Favier *et al.*, 2009] Aurélie Favier, Simon de Givry, and Philippe Jégou. Exploiting Problem Structure for Solution Counting. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP 09)*, volume 5732 of *LNCS*, pages 335–343, 2009.

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

[Freuder and Quinn, 1987] Eugene C. Freuder and Michael J. Quinn. The Use of Lineal Spanning Trees to Represent Constraint Satisfaction Problems. Technical Report 87-41, University of New Hampshire, 1987.

[Freuder and Wallace, 1991] Eugene C. Freuder and Richard J. Wallace. Selective Relaxation For Constraint Satisfaction Problems. In *Proceedings of the IEEE* 3^{rd} *International Conference on Tools with Artificial Intelligence*, pages 332–339, 1991.

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

[Fulkerson and Gross, 1965] D. R. Fulkerson and O. A. Gross. Incidence Matrices and Interval Graphs. *Pacific Journal of Mathematics*, 15 (3):835–855, 1965.

[Geschwender *et al.*, 2013] Daniel Geschwender, Shant Karakashian, Robert Woodward, Berthe Y. Choueiry, and Stephen D. Scott. Selecting the Appropriate Consistency Algorithm for CSPs Using Machine Learning Techniques. In *Pre-PhD Student Abstract and Poster Program, Proceedings of the 27th Conference on Artificial Intelligence (AAAI 2013)*, pages 1611–1612, 2013.

[Geschwender *et al.*, 2016] Daniel J. Geschwender, Robert J. Woodward, Berthe Y. Choueiry, and Stephen D. Scott. A Portfolio Approach for Enforcing Minimality in a Tree Decomposition. In *Doctoral Program of the International Conference on Principles and Practice of Constraint Programming (CP 2016)*, pages 1–10, 2016.

[Gogate and Dechter, 2008] Vibhav Gogate and Rina Dechter. Approximate Solution Sampling (and Counting) on AND/OR Spaces. In *Proceedings of* 14 *th International Conference on Principle and Practice of Constraint Programming (CP'08)*, volume 5202 of *LNCS*, pages 534–538. Springer, 2008.

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

[Gottlob *et al.*, 2000] Georg Gottlob, Nicola Leone, and Francesco Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124(2):243–282, 2000.

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

[Haralick and Elliott, 1980] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

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

[Horton, 1987] Joseph D. Horton. A Polynomial-Time Algorithm to Find the Shortest Cycle Basis of a Graph. *SIAM Journal on Computing*, 16(2):358–366, 1987.

[Howell *et al.*, 2018a] Ian Howell, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Solving Sudoku with Consistency: A Visual and Interactive Approach. In *Proceedings of the 27 th International Joint Conference on Artificial Intelligence*, pages 5829–5831, Stockholm, Sweden, 2018.

[Howell *et al.*, 2018b] Ian Howell, Robert J. Woodward, Berthe Y. Choueiry, and Hongfeng Yu. A Qualitative Analysis of Search Behavior: A Visual Approach. In *Proceedings of the 2 nd Workshop on Explainable Artificial Intelligence*, pages 65–71, Stockholm, Sweden, 2018.

[Janssen *et al.*, 1989] P. Janssen, Philippe Jégou, B. Nougier, and M.C. 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.

[Järvisalo *et al.*, 2012] Matti Järvisalo, Marijn J. H. Heule, and Armin Biere. Inprocessing Rules. In *Automated Reasoning: 6th International Joint Conference (IJCAR 2012)*, volume 7364 of *LNCS*, pages 355–370. Springer, 2012.

[Jeavons *et al.*, 1994] Peter G. Jeavons, David A. Cohen, and Marc Gyssens. A Structural Decomposition for Hypergraphs. *Contemporary Mathematics*, 178:161–177, 1994.

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

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

[Kadioglu *et al.*, 2011] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Selection and Scheduling. In *Proceedings of* 17 *th International Conference on Principle and Practice of Constraint Programming (CP'11)*, volume 6876 of *LNCS*, pages 454–469. Springer, 2011.

[Karakashian *et al.*, 2010] Shant Karakashian, Robert Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proceedings of AAAI-2010*, pages 101–107, Atlanta, GA, 2010.

[Karakashian *et al.*, 2013] 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 AAAI-2013*, pages 466–473, Bellevue, WA, 2013.

[Kavitha *et al.*, 2007] Telikepalli Kavitha, Kurt Mehlhorn, and Dimitrios Michail. New Approximation Algorithms for Minimum Cycle Bases of Graphs. In *Symposium on Theoretical Aspects of Computer Science (STACS 2007)*, volume 4393 of *LNCS*, pages 512–523, 2007.

[Kjærulff, 1990] U. Kjærulff. Triagulation of Graphs - Algorithms Giving Small Total State Space. Research Report R-90-09, Aalborg University, Denmark, 1990.

[Lecoutre *et al.*, 2007] Christophe Lecoutre, Stéphane Cardon, and Julien Vion. Conservative Dual Consistency. In *Proceedings of AAAI-2007*, pages 237–242, 2007.

[Lecoutre *et al.*, 2011] Christophe Lecoutre, Stéphane Cardon, and Julien Vion. Second-Order Consistencies. *JAIR*, 40:175–219, 2011.

[Lecoutre *et al.*, 2012] 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.

[Lecoutre *et al.*, 2013] Christophe Lecoutre, Anastasia Paparrizou, and Kostas Stergiou. Extending STR to a Higher-Order Consistency. In *Proceedings of AAAI-2013*, pages 576–582, Bellevue, WA, 2013.

[Lecoutre, 2009] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE Ltd & Wiley Press, 2009.

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

[Lim *et al.*, 2004] Ryan Lim, Venkata Praveen Guddeti, and Berthe Y. Choueiry. An Interactive System for Hiring and Managing Graduate Teaching Assistants. In *Conference on Prestigious Applications of Intelligent Systems (ECAI 04)*, pages 730–734, Valencia, Spain, 2004.

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

[Marinescu and Dechter, 2006] Radu Marinescu and Rina Dechter. Memory Intensive Branch-and-Bound Search for Graphical Models. In *Proceedings of the Twenty-*

First National Conference on Artificial Intelligence (AAAI 2006), pages 1200–1205, 2006.

[Mehlhorn and Michail, 2009] Kurt Mehlhorn and Dimitrios Michail. Minimum Cycle Bases: Faster and Simpler. *ACM Trans. Algorithms*, 6(1):1–13, December 2009.

[Michel and Van Hentenryck, 2012] Laurent Michel and Pascal Van Hentenryck. Activity-Based Search for Black-Box Constraint Programming Solvers. In *Proc. of CPAIOR 2012*, volume 7298, pages 228–243. Spring, 2012.

[Mohr and Masini, 1988] Roger Mohr and Gérald Masini. Good Old Discrete Relaxation. In *Proceedings of the Eigthth European Conference on Artificial Intelligence*, pages 651–656, Munich, Germany, 1988.

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

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

[Ostrowski *et al.*, 2002] Richard Ostrowski, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs. Recovering and Exploiting Structural Knowledge from CNF Formulas. In *Proceedings of* 8 th *International Conference on Principle and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, pages 185–199. Springer, 2002.

[Otten and Dechter, 2012] Lars Otten and Rina Dechter. Anytime AND/OR Depth-first Search for Combinatorial Optimization. *AI Communications*, 25(3):211–227, 2012.

[Palmieri *et al.*, 2016] Anthony Palmieri, Jean-Charles Régin, and Pierre Schaus. Parallel Strategies Selection. In *Proceedings of* 22 nd *International Conference on Principle and Practice of Constraint Programming (CP'16)*, volume 9892 of *LNCS*, pages 388–404. Springer, 2016.

[Paparrizou and Stergiou, 2012] Anastasia Paparrizou and Kostas Stergiou. Evaluating Simple Fully Automated Heuristics for Adaptive Constraint Propagation. In *Proceedings of the IEEE* 24 th *International Conference on Tools with Artificial Intelligence*, pages 880–885, 2012.

[Paparrizou and Stergiou, 2016] Anastasia Paparrizou and Kostas Stergiou. Strong Local Consistency Algorithms for Table Constraints. *Constraints*, 21(2):163–197, Apr 2016.

[Paparrizou and Stergiou, 2017] Anastasia Paparrizou and Kostas Stergiou. On Neighborhood Singleton Consistencies. In *Proceedings of the 26 th International Joint Conference on Artificial Intelligence*, pages 736–742, Melbourne, Australia, 2017.

[Pesant *et al.*, 2012] Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini. Counting-Based Search: Branching Heuristics for Constraint Satisfaction Problems. *JAIR*, 43:173–210, 2012.

[Planken *et al.*, 2008] Léon Planken, Mathijs de Weerdt, and Roman van der Krogt. P^{3}C: A New Algorithm for the Simple Temporal Problem. In *Proceedings of the 18th International Conference on Automated Planning & Scheduling (ICAPS 2008)*, pages 256–263, 2008.

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

[Reeson *et al.*, 2007] Christopher G. Reeson, Kai-Chen Huang, Kenneth M. Bayer, and Berthe Y. Choueiry. An Interactive Constraint-Based Approach to Sudoku. In *Proceedings of AAAI-2007*, pages 1976–1977, Vancouver, British Columbia, 2007.

[Reeson, 2016] Christopher G. Reeson. On Path Consistency for Binary Constraint Satisfaction Problems. Master's thesis, University of Nebraska-Lincoln, 2016.

[Refalo, 2004] Philippe Refalo. Impact-Based Search Strategies for Constraint Programming. In *Proceedings of* 10 th *International Conference on Principle and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, pages 557–571. Springer, 2004.

[Rish and Dechter, 2000] Irina Rish and Rina Dechter. Resolution versus Search: Two Strategies for SAT. *Journal of Automated Reasoning*, 24(1):225–275, Feb 2000.

[Rossi *et al.*, 1990] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the Equivalence of Constraint Satisfaction Problems. In *Proceedings of the Ninth European Conference on Artificial Intelligence*, pages 550–556, 1990.

[Rossi *et al.*, 2006] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

[Schulte *et al.*, 2015] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*, 2015.

[Schulte, 1996] Christian Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 477–478. Springer, 1996.

[Seidel, 1981] Raimund Seidel. A New Method for Solving Constraint Satisfaction Problems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 338–342, 1981.

[Shishmarev *et al.*, 2016] Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Visual Search Tree Profiling. *Constraints*, 21(1):77–94, 2016.

[Simonis and Aggoun, 2000] Helmut Simonis and Abder Aggoun. Search-Tree Visualisation. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, volume 1870 of *LNCS*, pages 191–208. Springer, 2000.

[Simonis and O'Sullivan, 2011] Helmut Simonis and Barry O'Sullivan. Almost Square Packing. In *Proceedings of* 8 th *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research Principle and Practice of Constraint Programming (CPAIOR 2011)*, volume 6697, pages 196–209. Springer, 2011.

[Simonis *et al.*, 2000] Helmut Simonis, Abder Aggoun, Nicolas Beldiceanu, and Eric Bourreau. Complex Constraint Abstraction: Global Constraint Visualisation. In *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, volume 1870 of *LNCS*, pages 299–317. Springer, 2000.

[Simonis *et al.*, 2010] Helmut Simonis, Paul Davern, Jacob Feldman, Deepak Mehta, Luis Quesada, and Mats Carlsson. A Generic Visualization Platform for CP. In *Proceedings of* 16 th *International Conference on Principle and Practice of Constraint Programming (CP'10)*, volume 6308 of *LNCS*, pages 460–474. Springer, 2010.

[Stergiou, 2008] Kostas Stergiou. Heuristics for Dynamically Adapting Propagation. In *Proceedings of the 18 th European Conference on Artificial Intelligence*, pages 485–489, 2008.

[Subbarayan and Pradhan, 2005] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non-increasing Variable Elimination Resolution for Preprocessing SAT Instances. In *Theory and Applications of Satisfiability Testing: 7th International Conference (SAT 2004)*, volume 3542 of *LNCS*, pages 276–291. Springer, 2005.

[Swearingn *et al.*, 2011] Amanda Swearingn, Berthe Y. Choueiry, and Eugene C. Freuder. A Reformulation Strategy for Multi-Dimensional CSPs: The Case Study of the SET Game. In *Ninth International Symposium on Abstraction, Reformulation and Approximation (SARA 2011)*, pages 107–116. AAAI Press, 2011.

[Tarjan and Yannakakis, 1984] Robert Endre Tarjan and Mihalis Yannakakis. Simple Linear-Time Algorithms to Test Chordality of Graphs, Test Acyclicity of Hypergraphs, and Selectively Reduce Acyclic Hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[Tsang, 1993] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London, UK, 1993.

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

[Valiant, 1979] Leslie G. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[Vion *et al.*, 2011] Julien Vion, Thierry Petit, and Narendra Jussien. Integrating Strong Local Consistencies into Constraint Solvers. In *14th Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2009*, volume 6080 of *LNAI*, pages 90–104. Springer, 2011.

[Wallace and Freuder, 1992] Richard J. Wallace and Eugene C. Freuder. Ordering Heuristics for Arc Consistency Algorithms. In *AI/GI/VI 92*, pages 163–169, 1992.

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

[Walsh, 1999] Toby Walsh. Search in a Small World. In *Proceedings of the 16 th International Joint Conference on Artificial Intelligence*, pages 1172–1177, Stockholm, Sweden, 1999.

[Wilcoxon, 1945] Frank Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[Woodward and Choueiry, 2017] Robert J. Woodward and Berthe Y. Choueiry. Weight-Based Variable Ordering in the Context of High-Level Consistencies. *ArXiv e-prints*, November 2017.

[Woodward *et al.*, 2011a] Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Neighborhood Inverse Consistency as Lookahead for Non-Binary CSPs. In *Proceedings of AAAI-2011*, pages 1830–1831, San Francisco, CA, 2011.

[Woodward *et al.*, 2011b] Robert Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Solving Difficult CSPs with Relational Neighborhood Inverse Consistency. In *Proceedings of AAAI-2011*, pages 112–119, San Francisco, CA, 2011.

[Woodward *et al.*, 2011c] 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.

[Woodward *et al.*, 2012] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Revisiting Neighborhood Inverse Consistency on Binary CSPs. In *Proceedings of* 18 *th International Conference on Principle and Practice of Constraint Programming (CP'12)*, volume 7514 of *LNCS*, pages 688–703. Springer, 2012.

[Woodward *et al.*, 2014] Robert J. Woodward, Anthony Schneider, Berthe Y. Choueiry, and Christian Bessiere. Adaptive Parameterized Consistency for Non-Binary CSPs by Counting Supports. In *Proceedings of* 20 *th International Conference on Principle and Practice of Constraint Programming (CP'14)*, volume 8656 of *LNCS*, pages 755–764. Springer, 2014.

[Woodward *et al.*, 2016a] Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Cycle-Based Singleton Local Consistencies. Technical Report TR-UNL-CSE-2016-0004, Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, 2016.

[Woodward *et al.*, 2016b] Robert J. Woodward, Shant Karakashian, Berthe Y. Choueiry, and Christian Bessiere. Witnessing Solution Counting in Tree-Structured Methods for CSPs. Technical Report TR-UNL-CSE-2016-0006, Department of

Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, 2016.

[Woodward *et al.*, 2017] Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessiere. Cycle-Based Singleton Local Consistencies. In *Proceedings of AAAI-2017*, pages 5005–5006, San Francisco, CA, 2017.

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

[Wotzlaw *et al.*, 2013] Andreas Wotzlaw, Alexander van der Grinten, and Ewald Speckenmeyer. Effectiveness of pre- and inprocessing for CDCL-based SAT solving. Technical report, Institut für Informatik, Universität zu Köln, Köln, Germany, 2013.

[Xu and Choueiry, 2003] Lin Xu and Berthe Y. Choueiry. A New Efficient Algorithm for Solving the Simple Temporal Problem. In Mark Reynolds and Abdul Sattar, editors, *10th International Symposium on Temporal Representation and Reasoning and Fourth International Conference on Temporal Logic (TIME-ICTL 03)*, pages 212–222. IEEE Computer Society Press, 2003.

[Xu *et al.*, 2008] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-Based Algorithm Selection for SAT. *JAIR*, 32:565–606, 2008.

[Yvars, 2008] Pierre-Alain Yvars. Using Constraint Satisfaction for Designing Mechanical Systems. *International Journal on Interactive Design and Manufcturing (IJIDeM)*, 2(3):161–167, 2008.

Abstract

Determining whether or not a Constraint Satisfaction Problem (CSP) has a solution is $\mathcal{NP}$-complete. CSPs are solved by inference (i.e., enforcing consistency), conditioning (i.e., doing search), or, more commonly, by interleaving the two mechanisms. The most common consistency property enforced during search is Generalized Arc Consistency (GAC). In recent years, new algorithms that enforce consistency properties stronger than GAC have been proposed and shown to be necessary to solve difficult problem instances.

We frame the question of balancing the cost and the pruning effectiveness of consistency algorithms as the question of determining *where*, *when*, and *how much* of a higher-level consistency to enforce during search. To answer the 'where' question, we exploit the topological structure of a problem instance and target high-level consistency where cycle structures appear. To answer the 'when' question, we propose a simple, reactive, and effective strategy that monitors the performance of backtrack search and triggers a higher-level consistency as search thrashes. Lastly, for the question of 'how much,' we monitor the amount of updates caused by propagation and interrupt the process before it reaches a fixpoint. Empirical evaluations on benchmark problems demonstrate the effectiveness of our strategies.

Résumé

Déterminer si un problème de satisfaction de contraintes (CSP) admet ou non une solution est $\mathcal{NP}$-complet. Les CSP sont résolus par inférence (c'est-à-dire, en appliquant un algorithme de cohérence), par énumeration (c'est-à-dire en effectuant une recherche avec retour sur trace ou backtracking), ou, plus souvent, en entrelaçant les deux mécanismes. La propriété de cohérence la plus couramment appliquée en cours du backtracking est GAC (Generalized Arc Consistency). Au cours des dernières années, de nouveaux algorithmes pour appliquer des cohérences plus fortes que GAC ont été proposés et se sont avérés nécessaires pour résoudre les problèmes difficiles.

Nous nous attaquons à la question de balancer d'une part le coût des algorithmes de cohérence et, d'autre part, leur pouvoir d'élagage et posons cette problématique comme étant celle de déterminer *où, quand, et combien* une cohérence doit-elle être appliquée en cours de backtracking. Pour répondre à la question « où », nous exploitons la structure topologique d'une instance du problème et focalisons la cohérence forte là où des structures cycliques apparaissent. Pour répondre à la question « quand », nous proposons une stratégie simple, réactive et efficace qui surveille la performance du backtracking puis déclenche une cohérence forte lorsque le nombre des pas de backtracking devient alarmant. Enfin, pour la question du « combien », nous surveillons les mises à jour provoquées par la propagation des contraintes et interrompons le processus dès qu'il devient inactif ou coûteux même avant qu'il n'atteigne un point fixe. Des évaluations empiriques sur des problèmes de référence établissent l'efficacité de nos stratégies.