EXTENDING THE CHOCO CONSTRAINT-SOLVER WITH HIGH-LEVEL CONSISTENCY WITH EVALUATION ON THE NONOGRAM PUZZLE

by

Khang Phan

AN UNDERGRADUATE THESIS

Presented to the Faculty of

The College of Art and Sciences at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Bachelor of Science

Major: Computer Science

Under the Supervision of Professor Berthe Y. Choueiry (advisor) and
Professor Mohammad Rashedul Hasan (co-advisor)

Lincoln, Nebraska

January, 2020

EXTENDING THE CHOCO CONSTRAINT-SOLVER WITH HIGH-LEVEL CONSISTENCY WITH EVALUATION ON THE NONOGRAM PUZZLE

Khang Phan, B.S.

University of Nebraska, 2020

Adviser: Berthe Y. Choueiry (adviser) and Mohammad Rashedul Hasan (co-adviser)

The Nonogram puzzle can be naturally modeled as a Constraint Satisfaction Problem (CSP) and advantageously solved by backtrack search coupled with techniques for enforcing and propagating consistency. Choco is a widely used constraint solver that consistently wins solver competitions in multiple categories. In this thesis, we extend the Choco solver with high-level consistency algorithms, namely SAC-1 and POAC-1. Then, we investigate the performance of three different constraint models of the Nonogram puzzle with Choco and three consistency algorithms, namely, GAC, SAC-1, and POAC-1.

Empirical evaluation on a benchmark instances of the Nonogram puzzle uncover the advantageous of high-level consistency algorithms: They can solve many instances in a backtrack-free manner and are able to solve the hardest instance, which GAC is unable to solve. On the easy instances, GAC remains competitive in terms of CPU time.

Table of Contents

List of Figures

List of Tables

ACKNOWLEDGMENTS

I would like to thank my adviser, Dr. Berthe Y. Choueiry, for her guidance and advice during my two years at the Constraint Systems Laboratory. I appreciate the the opportunity of working on several research projects in the lab, the conversations that we had about research, and her constructive feedback in the revision of this thesis. I would also like to thank my co-adviser, Dr. Mohammad Rashedul Hasan, for teaching the foundations of discrete mathematics and machine learning. I feel honored to have both of them as my advisers

I would like to thank Dr. Charles Prud'homme, Dr. Jean-Guillaume Fages, and other developers of the Choco constraint solver for providing a wonderful tool to study Constraint Programming. The insights that Dr. Prud'homme shared with me as well as his guidance and feedback have allowed me to improve my extension to Choco.

I wish to acknowledge the help provided by the people in the Constraint Systems Laboratory, including Mr. Ian Howell and Mr. Trieu Hung Tran. Trieu Hung conducted a similar research using the Stampede solver, the research solver in development at the Constraint Systems Laboratory. Trieu Hung has also provided the ternary-table constraint model for Nonogram instances and helped me verify my results. Ian is building the visualizing tool for constraint solvers that I use to produce many graphics in my research. He also provided generous constructive feedback on my Choco extension implementation.

This research was supported by a UCARE award from the University of Nebraska-Lincoln and by NSF grant RI-1619344. Experiments were conducted on the equipment at the Holland Computing Center at UNL.

Chapter 1

Introduction

In this thesis, we study the performance of solving three constraint models of the Nonogram puzzle with the constraint solver Choco. In this chapter, we present our motivation and contribution and discuss the organization of the thesis.

1.1 Motivation

Constraint Programming is a flexible and powerful paradigm for modeling and solving gubernatorial decision and optimization problems. Its practical applications include resource allocation, scheduling, and radio-frequency assignment.

Puzzles are whimsical problems that can advantageously be used to illustrate the usefulness and operations of Constraint Programming. They can also be used as the benchmark for comparing algorithms and as teaching tools in constraint modeling and algorithm design. Puzzles such as Minesweeper [Bayer *et al.*, 2006], Sudoku [Howell *et al.*, 2018a; Reeson *et al.*, 2007], and the Game of Set [Swearingn *et al.*, 2011] have successfully been used to this end. In this thesis, we investigate the performance of the Choco constraint-solver for solving the Nonogram puzzle. In particular, we enhance this solver with high-level consistency algorithms and evaluate the performance of Choco, which is based on two-way branching, in this context (i.e., Nonogram puzzle,

search with two-way branching, and high-level consistency).

1.2 Contribution

In this study, we extend Choco solver with high-level consistency algorithms and evaluate the performance of our extension in different models of the Nonogram puzzles. The high-level consistency algorithms we implement for Choco are SAC-1 [Debruyne and Bessière, 1997] and POAC-1 [Balafrej *et al.*, 2014].

1.3 Thesis Organization

This thesis is organized as follows. In Chapter 2, we introduce Constraint Satisfaction Problems (CSPs) and discuss various algorithms for solving them. We also introduce the Nonogram puzzle. In Chapter 3, we discuss Choco constraint-solver and how we extend it to include two high-level consistency algorithms, namely SAC-1 and POAC-1. In Chapter 4, we describe our experimental setup, our experiments, and results. Finally, in Chapter 5, we conclude this thesis and discuss possible future works.

Chapter 2

Background

In this chapter, we define a Constraint Satisfaction Problem and review the methods for solving it. Then, we introduce the Nonogram puzzle and how we describe the three constraint models that we use for solving it.

2.1 Constraint Satisfaction Problem

A Constraint Satisfaction Problem (CSP) is defined as follows. Given $P = (V, D, C)$ where

- $V = \{v_1, \cdots, v_n\}$ is a set of variables.
- D is the domain set of values such that $dom(v_i) \subseteq D$ for all $v_i \in V$. $dom(v_i)$ is the domain of v_i, that, the set of values that can be assigned to v_i.
- $C = \{c_1, \cdots, c_m\}$ is the set of constraints that restrict the allowed assignments of values to variables. Each constraint c_i is defined by its scope, $scp(c_i) \subseteq V$, and a relation $rel(c_i) \subseteq \prod_{v_j \in scp(c_i)} dom(v_j)$.

A solution to the CSP is an assignment of a value to each variable such that all the constraints are satisfied. The goal is usually to determine whether a solution exists (satisfaction) or to find a solution to the CSP (exemplification).

2.2 Algorithms for Solving CSPs

Determining whether or not a CSP is satisfiable is known to be NP-complete. Backtrack search is the only sound and complete algorithm for finding a solution to the CSP. To reduce the exponential number of combinations that backtrack search has to explore, we interleave, with search, the application of consistency algorithms, which remove values from the domains of the variables and/or tuples from the relations of the constraints that cannot appear in a solution to the CSP. Such operations reduce the size of the search space explored by the backtrack search.

Consistency algorithms enforce consistency properties. Below, we review the local consistency properties that are used in this thesis.

- Generalized arc-consistency (GAC) [Mackworth, 1977]: a CSP is generalized arc-consistent if, for every constraint, a value in the domain of a variable in the constraint can be extended to other variables in the same constraint. Various algorithms for enforcing GAC exist. In this thesis, we use the default GAC algorithms for the table constraint and the regular constraint that are available in Choco.

- Singleton arc-consistency (SAC) [Debruyne and Bessière, 1997]: Let a CSP with the domain of variable v_i reduced to value set $\{x_i\}$ be denoted as $P|_{v_i=x_i}$, then a CSP is singleton arc-consistent [Debruyne and Bessière, 1997] if for every variable $v_i \in V$ and value $x_i \in dom(v_i)$, $P|_{v_i=x_i}$ is arc consistent. SAC algorithm is the algorithm that enforces SAC property on the problem. A problem P after successfully enforced SAC is denoted as SAC(P). In this thesis, we implement in Choco the SAC-1 algorithm [Debruyne and Bessière, 1997].

- Partition-one arc-consistency (POAC) [Bennaceur and Affane, 2001] : a CSP P

is partition-one arc-consistent if it is singleton arc-consistent and for every value $x \in dom(v_i)$, then $x \in dom(v_i)$ of SAC($P|_{v_j=y}$) for every variable $v_j \neq v_i$ and value $y \in dom(v_j)$. Partition-one arc-consistency algorithm is the algorithm that enforces POAC property on the problem. In this thesis, we implement in Choco the POAC-1 algorithm [Balafrej *et al.*, 2014].

GAC is the standard consistency enforced in search. SAC and POAC are said to be *high-level consistency* (HLC) because, relatively to GAC, they never delete fewer values than GAC does. Further, POAC is known to be strictly stronger than SAC because it never filters out fewer values than SAC does [Balafrej *et al.*, 2014].

2.3 Nonogram

A Nonogram puzzle consists of two parts:

- A board of $n \times m$ cells that can either be colored black or left blank. Each row is numbered from 1 to n and each column is numbered from 1 to m. The cell at row i and column j is denoted $C_{i,j}$
- A set of n labels for the rows and m labels for the columns, where each label is given by $L = [l_1, l_2, \ldots, l_k]$ and $\forall 1 \leq i \leq k, l_i \in \mathbb{N}^+$ denotes the number of contiguous cells that are colored in black.

A solution to the puzzle is a coloration of the cells that are consistent with the labels of the rows and the columns.

The Nonogram puzzle is known to be NP-Complete [Ueda and Nagao, 1996].

Figure 2.1 shows a simple instance of the Nonogram puzzle and its solution.

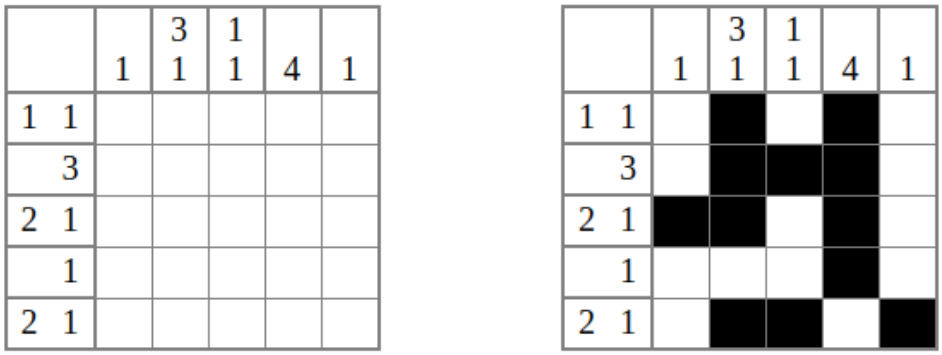

Figure 2.1: A simple Nonogram puzzle and its solution

2.4 Three Constraint Models for the Nonogram

In the CSP model of the Nonogram puzzle, each cell is represented by a Boolean variable, whose domain {0,1} indicates that the cell is blank or colored, respectively. Constraints are defined over the variables in the same rows or columns. We distinguish three constraint models for the Nonogram depending on the type of constraints used.

2.4.1 Global-table constraint model

In this model, each global-table constraint represents a label of a row or column. The constraint lists all *support* tuples, i.e., colorations that are consistent with the label. This model is memory consuming because the number of colorations is in factorial magnitude. In our previous example, the third row with a label of $L = [2, 1]$ can be expressed as a global-table constraint with the support list:

$$\{(1, 1, 0, 1, 0), (1, 1, 0, 0, 1), (0, 1, 1, 0, 1)\}$$

2.4.2 Regular constraint model

In this model, each regular constraint represents a label of row or column. The constrain is defined by a regular expression, where its variable sequence is the cells in

the row or column and the *deterministic finite automaton* (DFA) describes the label [Pesant, 2004]. In our previous Nonogram example, for the constraint for the third row, the sequence is the cells in that row and the FDA is described in Figure 2.2.

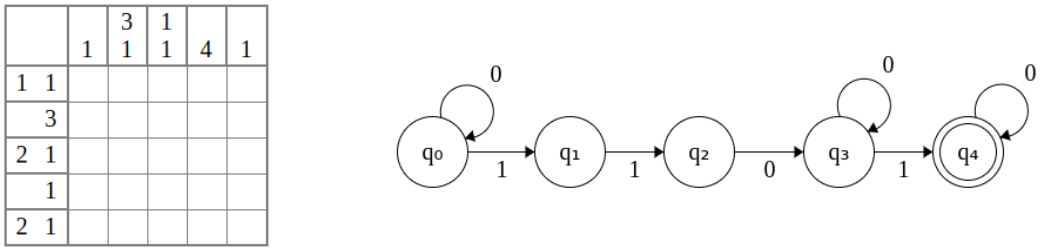

Figure 2.2: DFA for label $L = [2, 1]$

2.4.3 Ternary-table constraint model

In this model, each regular constraint is reformulated as an equivalent set of ternary-table constraints [Bessière *et al.*, 2008]. This operation adds a set of auxiliary variables to the model in each row and column as illustrated in Figure 2.3.

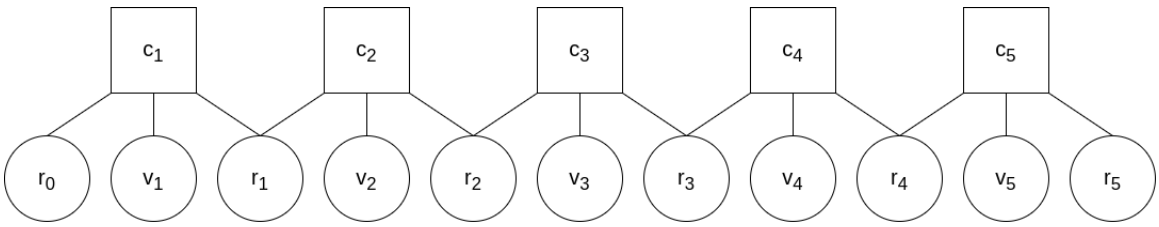

Figure 2.3: Ternary-table constraints network for the third row

In the Nonogram example of Figure 2.1, the regular constraint for the third row is replaced with the constraint network shown in Figure 2.3. In the network, $v_1, \cdots, v_5$ are the original variables, $r_0, \cdots, r_5$ are auxiliary variables, and $c_1, \cdots, c_5$ are the constraints. The domain of auxiliary variables and the definition of the constraints are shown in Table 2.1 and Table 2.2.

Table 2.1: Domain of auxiliary variables in the constraint network for the third row

Variable	Domain
r_0	$\{q_0\}$
r_1	$\{q_0, q_1, q_2, q_3, q_4\}$
r_2	$\{q_0, q_1, q_2, q_3, q_4\}$
r_3	$\{q_0, q_1, q_2, q_3, q_4\}$
r_4	$\{q_0, q_1, q_2, q_3, q_4\}$
r_0	$\{q_4\}$

Table 2.2: Definition of the ternary-table constraints for the third row

Constraint	Scope	Definition
c_1	r_0, v_1, r_1	$\{(q_0, 0, q_0), (q_0, 1, q_1)\}$
c_3	r_2, v_3, r_3	$\{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 1, q_2), (q_2, 0, q_3), (q_3, 0, q_3), (q_3, 1, q_4), (q_4, 0, q_4)\}$
c_3	r_2, v_3, r_3	$\{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 1, q_2), (q_2, 0, q_3), (q_3, 0, q_3), (q_3, 1, q_4), (q_4, 0, q_4)\}$
c_4	r_3, v_4, r_4	$\{(q_0, 0, q_0), (q_0, 1, q_1), (q_1, 1, q_2), (q_2, 0, q_3), (q_3, 0, q_3), (q_3, 1, q_4), (q_4, 0, q_4)\}$
c_5	r_4, v_5, r_5	$\{(q_3, 1, q_4), (q_4, 0, q_4)\}$

Summary

In this chapter, we introduced the definition of a Constraint Satisfaction Problem and reviewed the three consistency properties that are enforced during search, namely GAC, SAC-1, and POAC-1. We also introduced the Nonogram puzzle and its three constraint models that are based on global-table constraints, regular constraints, and ternary-table constraints, respectively.

Chapter 3

High-Levels Consistency Algorithms in Choco

In this chapter, we explain our implementations of the consistency algorithms SAC-1 [Debruyne and Bessière, 1997] and POAC-1 [Balafrej *et al.*, 2014] in Choco [Jussien *et al.*, 2008].

3.1 The Choco Constraint-Solver

In order to implement the SAC-1 and POAC-1 algorithms, we need to use three main actions of Choco:

- Extend: this function calculates the next decision, which is a variable-value pair.
- Propagate: this function applies the decision and then runs the chosen consistency algorithm.
- Repair: this function undoes the effects of the action Propagatex and returns the search to its state before propagation.

In our implementation, the three actions above are implemented as *CHOCO_assign*, *CHOCO_propagate*, and *CHOCO_backtrack*, respectively. The detailed implementation of these three actions in Java is shown in Algorithm 1.

Algorithm 1: Calls to CHOCO

```
1 Procedure CHOCO_assign(variable, value)
2     decision ← solver.getDecisionPath().makeIntDecision(variable,
        DecisionOperatorFactory.makeIntEq(), value)
3     solver.getDecisionPath().pushDecision(decision)
4     solver.getEnvironment().worldPush()
5     solver.getDecisionPath().buildNext()
6     solver.getObjectiveManager().postDynamicCut()
7     solver.getDecisionPath().apply()
1 Procedure CHOCO_propagate()
2     solver.getEngine().propagate()
1 Procedure CHOCO_backtrack()
2     solver.getEnvironment().worldPop()
3     solver.getDecisionPath().synchronize()
4     solver.getDecisionPath().buildNext()
5     solver.getDecisionPath().apply()
```

3.2 Singleton Arc Consistency (SAC-1)

We adapt the SAC-1 algorithm of Debruyne and Bessière[1997] to CHOCO as shown in the pseudocode of Algorithm 2.

This pseudocode requires the use of a circular queue, which we implement with an array whose last position links to the first position.

Algorithm 2: Implementation of the SAC-1 algorithm in Choco

```
1  Add variables to circular queue Q
2  queueHead ← random element in Q
3  queueEnd ← queueHead
4  repeat
5      currentVar ← queueHead
6      queueHead move forward
7      for x ∈ dom(currentVar) do
8          CHOCO_assign(currentVar, x)
9          CHOCO_propagate()
10         if contradiction occurs then
11             Remove x from dom(currentVar)
12             queueEnd ← currentVar
13         CHOCO_backtrack()
14     CHOCO_propagate()
15 until queueHead = queueEnd
```

3.3 Partition-One Arc Consistency (POAC-1)

To implement the POAC-1 algorithm of Balafrej *et al.* [2013], we define a new class called *VariableValueCounter*. This class has the following functions:

- It uses a *VariableMonitor* (provided by Choco) to get and save the list of variables whose domains have been updated during the singleton test.
- After each singleton test, if no contradictions occur, the class increases the counters of all variable-value pairs that have been removed during the singleton test.
- Finally, it can return the list of variable-value pairs that have been filtered in all singleton tests without contradiction by repeatedly calling function *next*.

Our pseudocode is given in Algorithm 3.

Algorithm 3: POAC-1 algorithm for Choco

```
1  Add variables to circular queue Q
2  queueHead ← random element in Q
3  queueEnd ← queueHead
4  FILL(vvpCounter, 0)
5  repeat
6      currentVar ← queueHead
7      queueHead move forward
8      counter ← 0
9      for x ∈ dom(currentVar) do
10         CHOCO_assign(currentVar, x)
11         CHOCO_propagate()
12         if contradiction occurs then
13             Remove x from dom(currentVar)
14             queueEnd ← currentVar
15         else
16             counter+ = 1
17             for (var, val) ∈ list of variable-value pairs removed do
18                 VariableValueCounter.increaseCounter((var, val))
19         CHOCO_backtrack()
20     for (var, val) ∈ VariableValueCounter.next() do
21         Remove val from dom(var)
22     CHOCO_propagate()
23 until queueHead = queueEnd
```

Summary

In this chapter, we presented our implementation of the SAC-1 and POAC-1 algorithm in Choco.

Chapter 4

Experimental Evaluation

In this chapter, we discuss our experimental set-up and our results.

4.1 Datasets

Our dataset consists of the Nonogram puzzles provided by the XCSP library of XCSP3.0 benchmark.[1] This library provides instances of Nonogram puzzles modeled with regular and global-table constraints. The ternary-table constraints are computed and provided by Tran [2019]

Our experiment runs the GAC, SAC-1, and POAC-1 algorithms on the following datasets:

1. 168 instances with global-table constraints.
2. 174 instances with regular constraints.
3. 174 instances with ternary-table constraint.

4.2 Environment

Our experiment is conducted on the Crane computer cluster provided by Holland

[1] `http://www.xcsp.org/series`

Computing Center. Each job is run on a single Intel Xeon E5-2670 2.60GHz CPU with 64GB RAM and a time limit of 4 hours per instance.

4.3 Summary of Results

For each algorithm and each constraint model, we report in Table 4.1 the following information:

1. The number of instances solved.
2. The number of instances solved in a backtrack-free manner.
3. The average CPU time per instance counting all instances. The character '>' indicates that at least one instance ran out of time.

Table 4.1: Performance of the three consistency algorithms on each constraint model

Model	# of instances	GAC	SAC-1	POAC-1
Number of instances solved				
Global table	168	168	168	168
Regular	174	173	174	174
Ternary table	174	173	174	174
Number of instances solved backtrack free				
Global table	168	24	163	163
Regular	174	24	168	168
Ternary table	174	21	172	172
Average CPU time for all instances (msec)				
Global table	-	237.80	214.33	240.69
Regular	-	>82,947.09	36,269.25	36,174.29
Ternary table	-	>83,138.82	4,205.07	5,131.22
Average CPU time for finished instances (msec)				
Global table	-	237.80	214.33	240.69
Regular	-	189.43	172.10	190.47
Ternary table	-	382.22	509.10	662.03

From the results shown in Table 4.1, we make the following observations:

- While the high-level consistency algorithms (i.e., SAC-1 and POAC-1) can solve all instances of all models, GAC does not terminate on one instance with the regular constraints and one instance with the ternary-table constraints.
- Both high-level consistency algorithms (i.e., SAC-1 and POAC-1) solve the most number of instances in a backtrack-free manner. Interestingly, both have the same performance.
- In terms of CPU time, the performance of the three algorithms differs. GAC shows the worst performance overall and fails to solve one instance of the regular-constraint model and one instance of the ternary-table constraint model. While SAC-1 is slightly better than POAC-1 on the models with the ternary-table constraints, their performances on the global-table constraint model and the regular-constraint model are indistinguishable.
- If we compare the models using average CPU time for finished instances, the regular-constraint model is the best model across all three algorithms, while the ternary-table constraint model has the worst running time. This is opposite what has been proposed by Bessière *et al.* [2008] that the ternary-table constraint model can be solved more efficiently than the regular-constraint model.

4.4 Runtime Distribution

To more closely analyze the CPU time performance of the three algorithms, we look at the cumulative curve of the runtime distribution, showing, as the time increases, the number of instances solved by each algorithm. These graphs are shown in Figure 4.1 (global-table constraint), Figure 4.2 (regular constraint), and Figure 4.3 (ternary-table constraint).

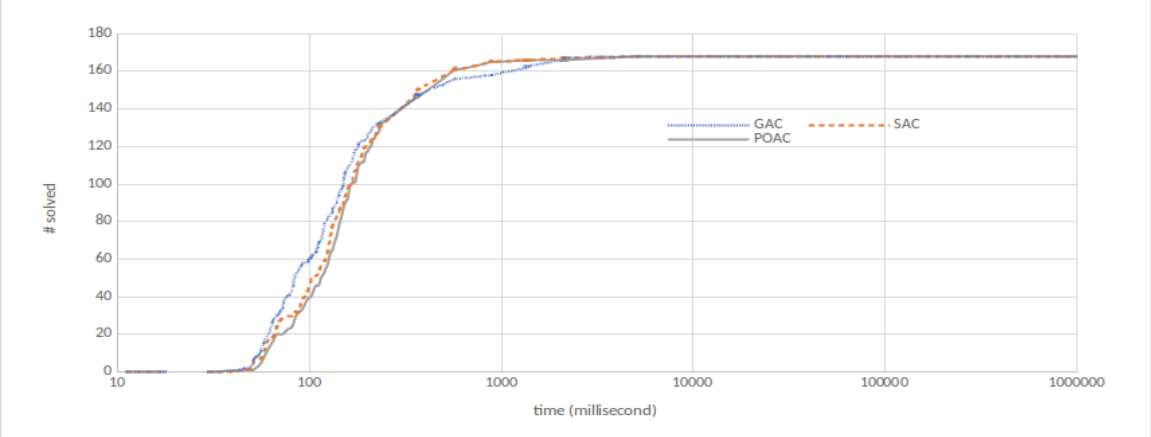

Figure 4.1: Runtime distribution for the model with global-table constraints

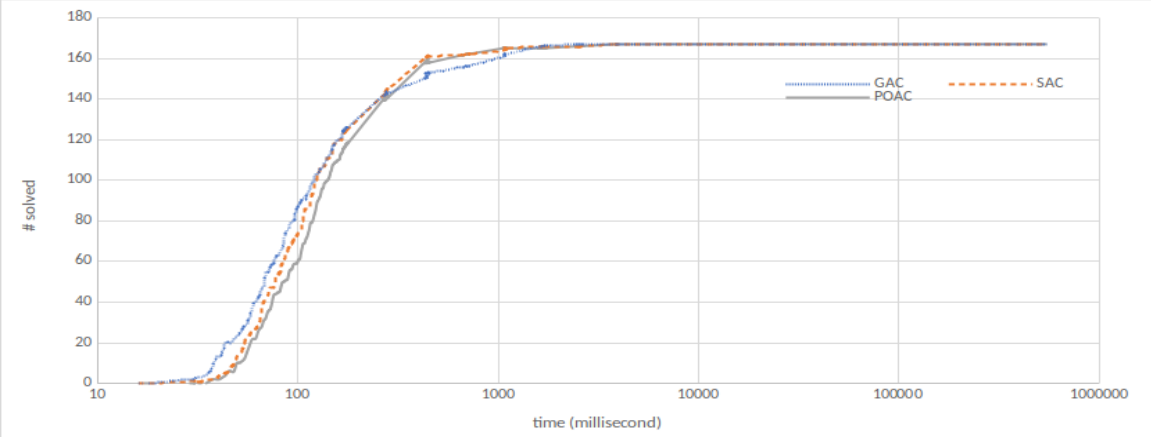

Figure 4.2: Runtime distribution for the model with regular constraints

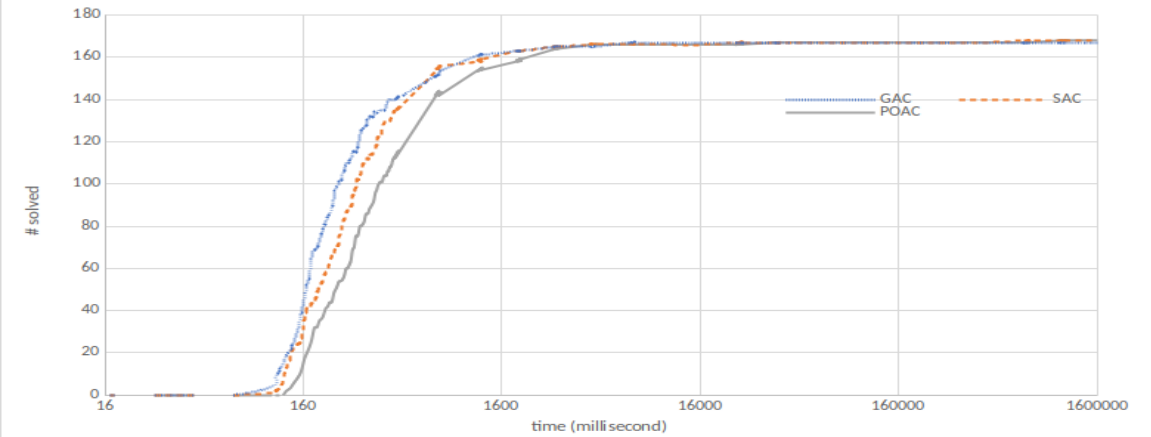

Figure 4.3: Runtime distribution for the model with ternary-table constraints

These graphs show that:

- GAC solves more instances than either SAC-1 and POAC-1 when the time limit is small (i.e., on easy instances). Further, SAC-1 slightly outperforms POAC-1.
- When the time limit is large (i.e., the puzzle is harder), all three algorithms seem to be equivalent.

4.5 Case Study of Instance #69

Instance #69 is the hardest of the 178 instances tested. Table 4.2 shows the detailed results of backtrack search on this instance for two of the constraint models knowing that the model for the global-table constraints is not available for Instance #69 because it is too large to store.

Table 4.2: Detailed search results for Instance #69

				MaxBpD	
Model	Algorithm	#BT	CPU time	Depth	Value
Regular	GAC	129,791,519	>14,400.03	58	1,593,321
	SAC-1	49,300,593	6,281.08	48	910,197
	POAC-1	49,300,593	6,261.37	48	910,197
Ternary	GAC	28,368,524	>14,400.03	41	256,532
	SAC-1	1,437,598	643.61	29	43,568
	POAC-1	1,437,598	778.30	29	43,568

This experiment on a difficult instance is interesting because it allows us to see the advantage of HLC. Indeed:

- This instance can be solved by SAC-1 and POAC-1 but not by GAC.

- For the ternary-table constraint model, SAC-1 is better than POAC-1, while for the regular-constraint model, POAC-1 is slightly better. The difference is very insignificant.
- The model with ternary-table constraints can be solved more efficiently than the model with the regular constraints as predicted by Bessière *et al.* [2008].

The charts *backtrack per depth* (BpD) graph [Howell *et al.*, 2018b] on Instance #69 are shown in Figure 4.4 for GAC, Figure 4.5 for SAC, and Figure 4.6 for POAC with the regular model on the left and the ternary-table model on the right. We notice that the higher-consistency algorithms (SAC and POAC versus GAC) can significantly reduce the backtracking effort (y-axis) and also shift the peak backtrack value to shallower depth levels.

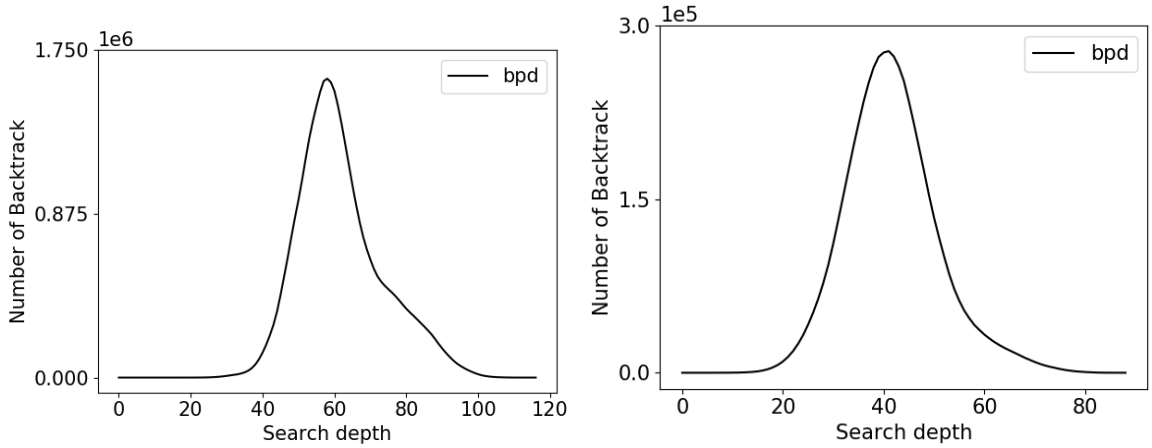

Figure 4.4: BpD of GAC on Instance #69: regular constraint (left) and ternary constraint (right)

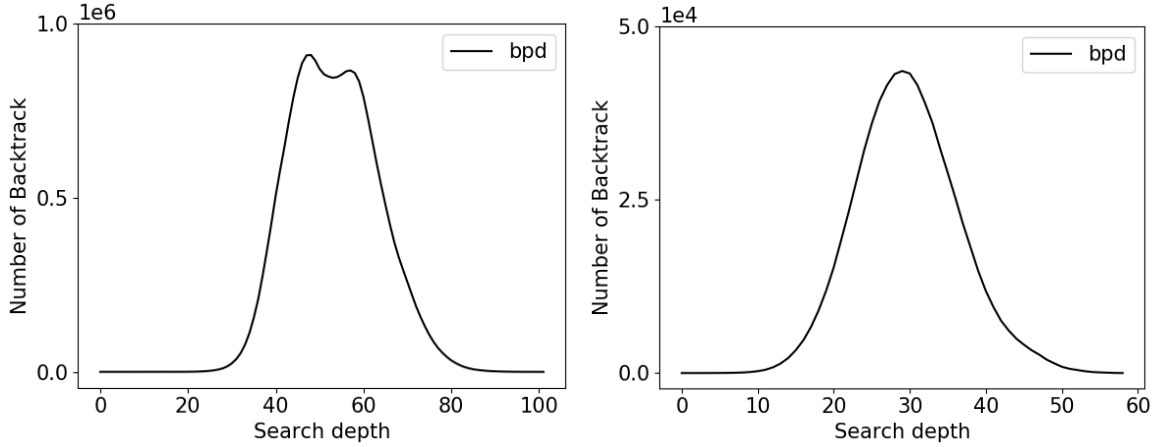

Figure 4.5: BpD of SAC on Instance #69: regular constraint (left) and ternary constraint (right)

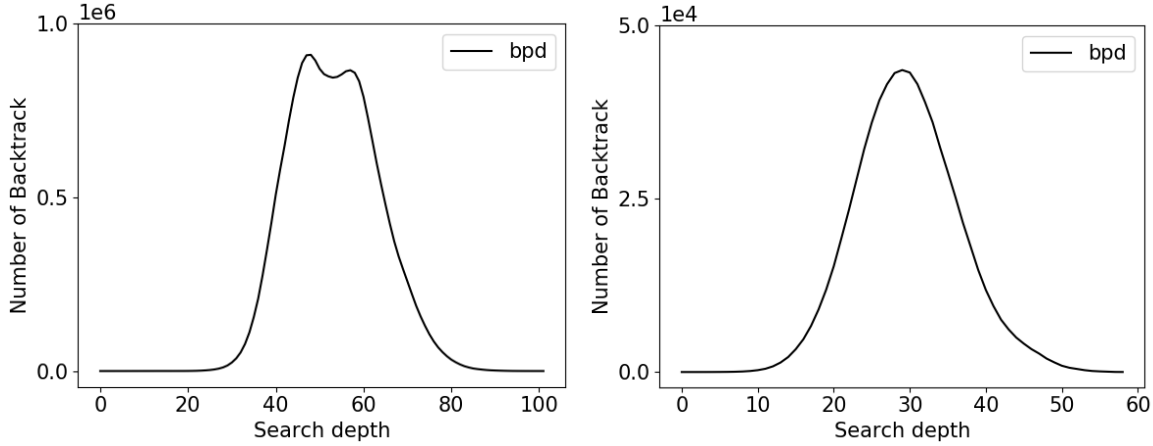

Figure 4.6: BpD of POAC on Instance #69: regular constraint (left) and ternary constraint (right)

Summary

In this chapter, we discussed our experiments and results. From the result, we noticed that in general, for small instances, GAC is slightly better than SAC-1 and POAC-1 in CPU time. However, as the instances were harder, SAC-1 and POAC-1 were more efficient in solving them.

Chapter 5

Conclusion and Future Work

In this chapter, we will conclude our study and suggest directions for future research

5.1 Conclusion

From our study, with the result we have from empirical evaluation, we conclude that:

- High-level consistency algorithms, namely SAC-1 and POAC-1, are more powerful than GAC because they can solve harder instances that GAC cannot.
- To achieve their efficiency, high-level consistency algorithms spend time for overhead work and preparation. In simple problems, the time spent for overhead work overwhelms the efficiency gained, thus increasing the total running time of the algorithms.

5.2 Future Work

Our experiments open up many directions for future research:

- Beside Nonogram and puzzles, many other CSP problems are available. We would evaluate our extension of Choco with different them and further improve our implementation.

- It would be very interesting to use alternative improvement of POAC-1, such as APOAC [Balafrej *et al.*, 2013], or trigger-based strategy, like PREPEAK [Woodward, 2018] to activate high-level consistency algorithms.

Bibliography

[Balafrej *et al.*, 2013] Amine Balafrej, Christian Bessière, Remi Coletta, and El-Houssine Bouyakhf. Adaptive Parameterized Consistency. In *Proceedings of* 19 th *International Conference on Principle and Practice of Constraint Programming (CP 2013)*, volume 8124 of *LNCS*, pages 143–158. Springer, 2013.

[Balafrej *et al.*, 2014] Amine Balafrej, Christian Bessière, El-Houssine Bouyakhf, and Gilles Trombettoni. Adaptive Singleton-Based Consistencies. In *Proceedings of AAAI-2014*, pages 2601–2607, 2014.

[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 *et al.*, 2008] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. SLIDE: A useful special case of the CARDPATH

constraint. In *Proceedings of* 18 *th European Conference on Artificial Intelligence (ECAI 2008)*, pages 475–479, 2008.

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

[Howell *et al.*, 2018a] Ian Howell, Robert J. Woodward, Berthe Y. Choueiry, and Christian Bessière. 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.

[Jussien *et al.*, 2008] Narendra Jussien, Guillaume Rochart, and Xavier Lorca. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)*, pages 1–10, Paris, France, France, 2008.

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

[Pesant, 2004] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of* 10 *th International Conference on Principles and Practice of Constraint Programming (CP 2004)*, volume 3258 of *LNCS*, pages 482–495. Springer, 2004.

[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.

[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.

[Tran, 2019] Trieu Hung Tran. Modeling and Solving the Nonogram Puzzle Using Constraint Programming, 2019.

[Ueda and Nagao, 1996] Nobuhisa Ueda and Tadaaki Nagao. NP-completeness Results for NONOGRAM via Parsimonious Reductions. Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan, 1996.

[Woodward, 2018] Robert J. Woodward. *Higher-Level Consistencies: Where, When, and How Much.* PhD thesis, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2018.