



NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

PROGRAM OF POSTGRADUATE STUDIES

PhD THESIS

Constraint Programming: Algorithms and Systems

Nikolaos I. Pothitos

ATHENS

JUNE 2022



ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

**Προγραμματισμός με Περιορισμούς:
Αλγόριθμοι και Συστήματα**

Νικόλαος Ι. Ποθητός

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2022

PhD THESIS

Constraint Programming: Algorithms and Systems

Nikolaos I. Pothitos

SUPERVISOR: Panagiotis Stamatopoulos, Assistant Professor UoA

THREE-MEMBER ADVISORY COMMITTEE:

Panagiotis Stamatopoulos, Assistant Professor UoA

Constantin Halatsis, Emeritus Professor UoA

Stavros Kolliopoulos, Professor UoA

SEVEN-MEMBER EXAMINATION COMMITTEE

(Signature)

(Signature)

Panagiotis Stamatopoulos,
Assistant Professor UoA

Constantin Halatsis,
Emeritus Professor UoA

(Signature)

(Signature)

Stavros Kolliopoulos,
Professor UoA

Manolis Koubarakis,
Professor UoA

(Signature)

(Signature)

Nick Bassiliades,
Professor AUTH

Ioannis Hatzilygeroudis,
Professor UPatras

(Signature)

Ilias Sakellariou,
Assistant Professor UoM

Examination Date 6/6/2022

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

Προγραμματισμός με Περιορισμούς: Αλγόριθμοι και Συστήματα

Νικόλαος Ι. Ποθητός

ΕΠΙΒΛΕΠΩΝ ΚΑΘΗΓΗΤΗΣ: Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής ΕΚΠΑ

ΤΡΙΜΕΛΗΣ ΕΠΙΤΡΟΠΗ ΠΑΡΑΚΟΛΟΥΘΗΣΗΣ:

Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής ΕΚΠΑ

Κωνσταντίνος Χαλάτσης, Ομότιμος Καθηγητής ΕΚΠΑ

Σταύρος Κολλιόπουλος, Καθηγητής ΕΚΠΑ

ΕΠΤΑΜΕΛΗΣ ΕΞΕΤΑΣΤΙΚΗ ΕΠΙΤΡΟΠΗ

(Υπογραφή)

(Υπογραφή)

**Παναγιώτης Σταματόπουλος,
Επίκουρος Καθηγητής ΕΚΠΑ**

**Κωνσταντίνος Χαλάτσης,
Ομότιμος Καθηγητής ΕΚΠΑ**

(Υπογραφή)

(Υπογραφή)

**Σταύρος Κολλιόπουλος,
Καθηγητής ΕΚΠΑ**

**Μανόλης Κουμπάρκης,
Καθηγητής ΕΚΠΑ**

(Υπογραφή)

(Υπογραφή)

**Νικόλαος Βασιλειάδης,
Καθηγητής ΑΠΘ**

**Ιωάννης Χατζηλυγερούδης,
Καθηγητής ΠΠ**

(Υπογραφή)

**Ηλίας Σακελλαρίου,
Επίκουρος Καθηγητής ΠΑΜΑΚ**

Ημερομηνία εξέτασης 6/6/2022

ABSTRACT

Constraint Programming aims at the easy declaration and fast resolution of Constraint Satisfaction Problems (CSPs) like course scheduling, radio link frequency assignment, etc. To solve the problems, Constraint Programming is based on

- search methods and
- constraint propagation.

This dissertation contributes on both. Specifically:

1. We develop novel search methods that are based on new heuristics. These new heuristics implement the *gradual* randomization of deterministic heuristics. We create hybrid heuristics that exploit the advantages of both deterministic and random heuristics.
2. We demonstrate how the MapReduce framework can be used for speeding up and distributing the search of a CSP solution to all the available solvers-workers.
3. We highlight the advantages of relaxed constraint propagation levels like *bounds consistency* in comparison to higher levels like *arc consistency*. We propose new relaxed constraint propagation levels, and we compare their performance to higher propagation levels, both in theory and practice. We answer the question about when it is worth to employ relaxed constraint propagation levels.

Our contributions were tested using mostly CSPs that occur in the real world and a wide range of CSPs included in official Constraint Programming solvers competitions. We used Naxos Solver as a practical open-source Constraint Programming solver to conduct our experiments.

SUBJECT AREA: Artificial Intelligence, Constraint Satisfaction

KEYWORDS: search, heuristics, randomization, MapReduce, constraint propagation, bounds consistency, maintaining arc consistency

ΠΕΡΙΛΗΨΗ

Ο Προγραμματισμός με Περιορισμούς (Constraint Programming) αποσκοπεί στην εύκολη διατύπωση και γρήγορη επίλυση των λεγόμενων Προβλημάτων Ικανοποίησης Περιορισμών (Constraint Satisfaction Problems – CSPs) όπως η κατάστρωση ωρολογίων προγραμμάτων, η ανάθεση συχνοτήτων σε ραδιοφωνικούς σταθμούς χωρίς παρεμβολές μεταξύ τους κ.ά. Για την επίλυση των προβλημάτων, ο Προγραμματισμός με Περιορισμούς βασίζεται

- στις μεθόδους αναζήτησης (search methods) και
- στη διάδοση περιορισμών (constraint propagation).

Η διατριβή αυτή συνεισφέρει και στους δύο αυτούς πυλώνες. Πιο συγκεκριμένα:

1. Αναπτύσσουμε καινούργιες μεθόδους αναζήτησης που βασίζονται σε καινοτόμους ευρετικούς κανόνες. Οι κανόνες αυτοί υλοποιούν τη *βαθμιαία* τυχαιοποίηση των ντετερμινιστικών ευρετικών κανόνων. Δημιουργούμε ένα υβρίδιο με στόχο την εκμετάλλευση των πλεονεκτημάτων τόσο των ντετερμινιστικών όσο και των τυχαίων ευρετικών κανόνων.
2. Αξιοποιούμε το πλαίσιο MapReduce προκειμένου να επιταχύνουμε και να κατανείμουμε την αναζήτηση λύσης ενός Προβλήματος Ικανοποίησης Περιορισμών σε όλους τους επιλυτές-εργάτες που τυχάνει να έχουμε στη διάθεσή μας.
3. Αναδεικνύουμε τα πλεονεκτήματα των χαλαρών επιπέδων διάδοσης περιορισμών, όπως η συνέπεια ορίων (bounds consistency) έναντι υψηλότερων επιπέδων όπως η συνέπεια ακμών (arc consistency). Προτείνουμε καινούργιες μορφές χαλαρών επιπέδων διάδοσης περιορισμών και συγκρίνουμε την απόδοσή τους σε σχέση με τα υψηλότερα επίπεδα διάδοσης περιορισμών, τόσο θεωρητικά όσο και πρακτικά. Απαντάμε στην ερώτηση για το πότε συμφέρει να χρησιμοποιούμε χαλαρά επίπεδα διάδοσης περιορισμών.

Οι συνεισφορές μας δοκιμάστηκαν ως επί το πλείστον σε Προβλήματα Ικανοποίησης Περιορισμών από τον πραγματικό κόσμο, αλλά και σε μια ευρύτερη γκάμα προβλημάτων που χρησιμοποιούνται σε επίσημους διαγωνισμούς επιλυτών Προγραμματισμού με Περιορισμούς. Ένας τέτοιος πρακτικός επιλυτής ανοικτού κώδικα είναι ο Naxos Solver που χρησιμοποιήσαμε ως το πεδίο εφαρμογής των πειραμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή Νοημοσύνη, Ικανοποίηση Περιορισμών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: αναζήτηση, ευρετικοί κανόνες, τυχαιοποίηση, MapReduce, διάδοση περιορισμών, συνέπεια ορίων, διατήρηση συνέπειας ακμών

To Jesus Christ "by Whom all things were made"

To Matina-Maria, Ioannis, Thomas, and Maria-Varvara

ACKNOWLEDGMENTS

*Aren't we too grown for games?
Aren't we too grown to play around?
Young enough to chase
But old enough to know better*

Maroon 5 – What lovers do ft. SZA

Back in the '90s when I was kid, we visited with my school the Geology Museum of the University of Athens. I still remember the awe I felt in front of the big buildings. At that time, it seemed to me like an elusive dream to be just an undergraduate student. I could not even imagine that a few decades later I would be defending my doctoral dissertation!

I am deeply grateful to Prof. Panagiotis Stamatopoulos for our strong and close cooperation for more than half of my life. Thanks for the empathy, generosity, motivation, support in scientific, technical, and personal level!

I would like to thank my Three-member Advisory Committee Professors Constantin Halatsis and Stavros Kolliopoulos and the Seven-member Examination Committee. Special thanks to Prof. Ilias Sakellariou for his numerous and valuable comments and corrections.

I express my most sincere appreciation to Prof. Isambo Karali and Dr. Anastasia Paparrizou for their advice throughout this journey. I also really appreciate the contribution of Ektoras Tavoularis who updated the Continuous Integration system of our Constraint Programming *Naxos Solver*.

Many thanks to Dr. Nikos Raptis and my “Nokian” colleagues for their support, fun, and encouragement they generously offered to me always!

I would like to thank Bodossaki Foundation and Ioannis Detsis, Georgios Bibilas, Ioannis Mathioudakis, Sotiris Laganopoulos, and Dr. Eleni Detsi for their scholarship during the dark years of the Greek crisis. Special thanks to Christos Karatsalos who informed me about the scholarship; without his encouragement I would never have applied.

I cannot forget the warm hospitality by Christos Palantzas during my stay in Lamia for the 7th Hellenic Artificial Intelligence conference.

I grab the opportunity to express my total support and solidarity with Dr. Timnit Gebru and Dr. Margaret Mitchell, the two leaders of Google Ethical AI research that were recently fired. We have a long way to go until corporate AI becomes ethical.

I would like to thank my parents Yannis and Elli and my sister Evi.

Many thanks to my wife Matina-Maria and my children Ioannis, Thomas, and Maria-Varvara! They all came into my life after beginning this dissertation, so I



Figure 1: Harold Cohen, *090921*, print

dedicate it to them! I love you!

Finally, I would like to thank the emeritus professor Harold Cohen (1928–2016) who sent to me two wonderful paintings, Fig. 1 and 1.1, designed using Artificial Intelligence. Just for the record, I quote the permission that he granted me. Rest in peace.

Date: Sat, 15 Oct 2011 09:05:11 -0400
From: Harold Cohen <hcohen@ucsd.edu>

Dear Mr. Pothitos,

Hope these will serve your needs. Please understand that they may not be used in any context other than your PhD thesis.

For captions: these are both prints; the "titles" are actually the dates in ymd format.

Best wishes,
Harold Cohen

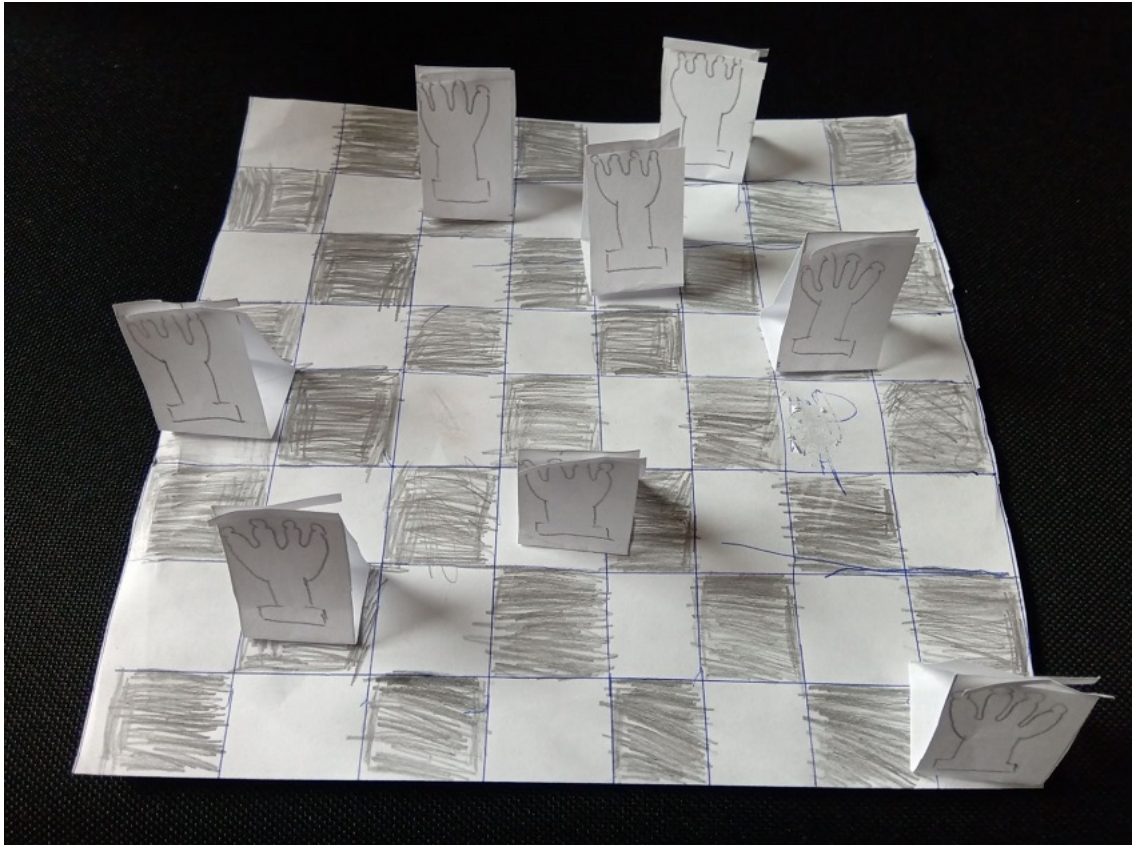


Figure 2: A solution to the eight queens CSP. The “do it yourself” chessboard has been created from paper with the precious help of my kids while staying at home after being infected with COVID-19.

LIST OF PUBLICATIONS

- P1. N. Pothitos and P. Stamatopoulos, "The dilemma between arc and bounds consistency," *International Journal of Intelligent Systems*, vol. 35, no. 10, pp. 1467–1491, 2020.
- P2. N. Pothitos and P. Stamatopoulos, "Building search methods with self-confidence in a constraint programming library," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 4, pp. 1860003, 1–34, 2018.
- P3. N. Pothitos and P. Stamatopoulos, "Piece of Pie Search: Confidently exploiting heuristics," in *SETN 2016*. New York: ACM, 2016, pp. 8:1–8:8.
- P4. N. Pothitos and P. Stamatopoulos, "Constraint Programming MapReduce'd," in *SETN 2016*. New York: ACM, 2016, pp. 5:1–5:4.
- P5. N. Pothitos, P. Stamatopoulos, and K. Zervoudakis, "Course scheduling in an adjustable constraint propagation schema," in *ICTAI 2012*, vol. 1. IEEE, 2012, pp. 335–343.
- P6. N. Pothitos, G. Kastrinis, and P. Stamatopoulos, "Constraint propagation as the core of local search," in *SETN 2012*, ser. LNCS (LNAI), vol. 7297. Heidelberg: Springer, 2012, pp. 9–16.
- P7. N. Pothitos and P. Stamatopoulos, "Flexible management of large-scale integer domains in CSPs," in *SETN 2010*, ser. LNCS (LNAI), vol. 6040. Heidelberg: Springer, 2010, pp. 405–410.

ΣΥΝΟΠΤΙΚΗ ΠΑΡΟΥΣΙΑΣΗ

Ο στόχος της διατριβής είναι να συνεισφέρει στην έρευνα πάνω στον Προγραμματισμό με Περιορισμούς (Constraint Programming) που είναι μία περιοχή της Τεχνητής Νοημοσύνης. Στο πλαίσιο αυτής της εργασίας, παρήχθησαν καινοτόμα θεωρητικά αποτελέσματα τα οποία είχαν ως πρακτική συνέπεια την επιτάχυνση της επίλυσης των Προβλημάτων Ικανοποίησης Περιορισμών.

Προβλήματα Ικανοποίησης Περιορισμών

Ένα παράδειγμα Προβλήματος Ικανοποίησης Περιορισμών (Constraint Satisfaction Problem – CSP) από τον πραγματικό κόσμο που επιλύεται μέσω Προγραμματισμού με Περιορισμούς, είναι η κατάστρωση ωρολογίων προγραμμάτων για εκπαιδευτικά ιδρύματα (course scheduling). Σε ένα τέτοιο πρόβλημα υπάρχουν αυστηροί περιορισμοί όπως το να μην διδάσκονται ταυτόχρονα δύο μαθήματα στην ίδια αίθουσα και να μην διδάσκονται ταυτόχρονα δύο μαθήματα του ίδιου έτους. Λιγότερο αυστηροί περιορισμοί αφορούν στη μη ύπαρξη κενών στα προσωπικά ωρολόγια προγράμματα φοιτητών και καθηγητών.

Άλλο παράδειγμα Προβλήματος Ικανοποίησης Περιορισμών είναι η κατανομή των ραδιοφωνικών συχνοτήτων σε μία χώρα, έτσι ώστε ο ένας ραδιοφωνικός σταθμός να μην κάνει παρεμβολές στον άλλον (radio link frequency assignment). Ένας αυστηρός περιορισμός σε αυτήν την περίπτωση είναι ότι δύο σταθμοί που εκπέμπουν στην ίδια περιοχή οφείλουν να χρησιμοποιούν συχνοότητες που διαφέρουν τουλάχιστον κατά ένα συγκεκριμένο αριθμό μεγακύκλων.

Οι συνεισφορές της διατριβής

Όπως θα παρουσιαστεί διεξοδικότερα παρακάτω, στον Προγραμματισμό με Περιορισμούς επιλύουμε τα Προβλήματα Ικανοποίησης Περιορισμών εναλλάσσοντας

- μία μέθοδο αναζήτησης (search method) και
- μία διαδικασία διάδοσης περιορισμών (constraint propagation).

Η συνεισφορά της διατριβής αφορά και στους δύο αυτούς άξονες και συγκεκριμένα στη

1. βαθμιαία τυχαιοποίηση ευρετικών κανόνων αναζήτησης [P2,P3]
2. κατανομή της αναζήτησης μέσω του πλαισίου MapReduce [P4]
3. ανάδειξη των πλεονεκτημάτων των χαλαρών διαδικασιών διάδοσης περιορισμών [P1,P5]

Πλαίσιο παρατηρήσεων και πειραμάτων

Η πρόκληση στα Προβλήματα Ικανοποίησης Περιορισμών έγκειται στη μεγάλη ποικιλία τους και στις διαφορές της δυσκολίας επίλυσής τους. Αναπόφευκτα, κάποιος που χρησιμοποιεί τον Προγραμματισμό με Περιορισμούς, δεν είναι δυνατόν να ασχοληθεί με όλα τα είδη προβλημάτων. Επιπλέον, η επινόηση και μελέτη ακόμα και τεχνητών Προβλημάτων Ικανοποίησης Περιορισμών είναι σύνηθες φαινόμενο στη βιβλιογραφία.

Τίθεται λοιπόν το ερώτημα: σε ποια προβλήματα οφείλει να δοκιμάσει τη μεθοδολογία που προτείνει ο ερευνητής ώστε να είναι όσο το δυνατόν αμερόληπτος; Είναι θεμιτό να ασχολείται με τεχνητά προβλήματα; Ακόμα και αν συμφωνήσουμε στην εξέταση ενός συγκεκριμένου προβλήματος, με ποιο τρόπο διατυπώνεται αυτό σε ένα σύστημα Προγραμματισμού με Περιορισμούς; Σημειώνεται ότι θεμελιώδης στόχος του Προγραμματισμού με Περιορισμούς είναι η ευκολία διατύπωσης.

Κατά τη διεξαγωγή της έρευνας της παρούσης διατριβής, επενδύσαμε στην επίλυση απαιτητικών προβλημάτων από τον πραγματικό κόσμο, όπως η κατάστρωση ωρολογίων προγραμμάτων για εκπαιδευτικά ιδρύματα. Ασχοληθήκαμε όμως και με ευρύτερα προβλήματα, τεχνητά και μη, όπως αυτά που χρησιμοποιούνται σε επίσημους διαγωνισμούς ταχύτητας συστημάτων Προγραμματισμού με Περιορισμούς. Και όλα αυτά επεκτείνοντας ένα σύστημα Προγραμματισμού με Περιορισμούς ανοικτού κώδικα Naxos Solver¹ που έχουμε δημιουργήσει οι ίδιοι και χρησιμοποιείται ευρέως λόγω της ευκολίας διατύπωσης Προβλημάτων Ικανοποίησης Περιορισμών που παρέχει.

1 Τυχαίοι και ντετερμινιστικοί ευρετικοί κανόνες: Γεφυρώνοντας το χάσμα

Ένα Πρόβλημα Ικανοποίησης Περιορισμών αποτελείται από ένα σύνολο *μεταβλητών* (variables) και ένα σύνολο *περιορισμών* (constraints) που τις συνδέουν. Κάθε μεταβλητή παίρνει τιμή από ένα πεπερασμένο σύνολο ακεραίων που ονομάζεται *πεδίο τιμών* της (finite domain). Ένας συνδυασμός τιμών των μεταβλητών που δεν παραβιάζει κανέναν περιορισμό είναι *λύση* του προβλήματος. Επομένως, για να βρούμε μία λύση, αρκεί να εξετάσουμε όλους τους δυνατούς συνδυασμούς τιμών των μεταβλητών του προβλήματος.

1.1 Η αναγκαιότητα των ευρετικών κανόνων

Ωστόσο, καθώς αυξάνονται οι μεταβλητές του προβλήματος, οι συνδυασμοί που πρέπει να εξετάσουμε αυξάνονται εκθετικά. Επειδή είναι απαγορευτικός ο αριθμός των υποψηφίων λύσεων για να τις εξετάσουμε όλες, χρειαζόμαστε τους λεγόμενους *ευρετικούς κανόνες* (heuristics) προκειμένου να εξετάσουμε πρώτα τις περισσότερο «υποσχόμενες» υποψήφίες λύσεις.

Αν υπήρχε τέλειος ευρετικός κανόνας, τότε θα καταργούταν η ανάγκη να χρησιμοποιήσουμε έναν αλγόριθμο αναζήτησης. Θα δίναμε όλες τις υποψήφίες

¹<https://github.com/pothitos/naxos>

λύσεις ως είσοδο στον ευρετικό κανόνα και εκείνος θα μας επέστρεφε απευθείας τη λύση. Όμως τέλειος ευρετικός κανόνας δεν υπάρχει.

Αυτό σημαίνει ότι, σε μία ακραία αλλά όχι σπάνια περίπτωση, ένας ευρετικός κανόνας μπορεί να δίνει προτεραιότητα σε κακές υποψήφια λύσεις, καθυστερώντας έτσι την αναζήτηση. Αυτός είναι ο λόγος για τον οποίο πολλές φορές η μέθοδος αναζήτησης επιλέγει στην τύχη την επόμενη υποψήφια λύση, αντί να χρησιμοποιήσει κάποιον ντετερμινιστικό ευρετικό κανόνα.

1.2 Βαθμιαία τυχαιοποίηση ευρετικών κανόνων

Ανάμεσα στα δύο άκρα των ντετερμινιστικών και τυχαίων ευρετικών κανόνων, στο πλαίσιο αυτής της διατριβής δημιουργήθηκε για πρώτη φορά η δυνατότητα *βαθμιαίας* τυχαιοποίησης ενός ντετερμινιστικού ευρετικού κανόνα. Κατασκευάστηκαν υβριδικοί ευρετικοί κανόνες που δοκιμάστηκαν σε δύσκολα προβλήματα ικανοποίησης περιορισμών από τον πραγματικό κόσμο.

Επιπλέον, δημιουργήθηκε μία καινούργια μέθοδος αναζήτησης PoPS (Piece of Pie Search) η οποία χρησιμοποιεί αποτελεσματικά τους παραπάνω υβριδικούς ευρετικούς κανόνες. Όσο λιγότερες αναθέσεις τιμών έχουν πραγματοποιηθεί, τόσο πιο τυχαίο είναι οι ευρετικοί κανόνες που χρησιμοποιούνται. Όσο περισσότερες αναθέσεις έχουν πραγματοποιηθεί, τόσο περισσότερο ντετερμινιστικοί ευρετικοί κανόνες χρησιμοποιούνται [P2,P3].

Μοιάζει με την περίπτωση που ξεκινάει μια παρτίδα σκάκι. Αρχικά, έχουμε περισσότερες επιλογές και μπορούμε να επιλέξουμε τυχαία μέσα από ένα μεγαλύτερο σύνολο κινήσεων. Όσο όμως ο χρόνος του παιχνιδιού κυλάει, η στρατηγική μας γίνεται ολοένα και περισσότερο ντετερμινιστική.

2 Ενσωμάτωση του MapReduce στον Προγραμματισμό με Περιορισμούς

Η πρόκληση των ημερών μας δεν είναι πάντοτε να περιορίζουμε έναν αλγόριθμο ώστε να απαιτεί λιγότερους υπολογιστικούς πόρους. Το ζητούμενο πολλές φορές πλέον είναι πώς θα αξιοποιήσουμε έναν «στράτο» από φτηνούς υπολογιστές-εργάτες που τίθενται εύκολα στις υπηρεσίες μας.

Για την περίπτωση του Προγραμματισμού με Περιορισμούς, ας φανταστούμε ότι έχουμε στη διάθεσή μας έναν μεγάλο αριθμό υπολογιστών στο νέφος (cloud) της Google ή της Amazon. Πώς μπορούμε να τους εκμεταλλευτούμε για να επιλύσουμε γρηγορότερα ένα Πρόβλημα Ικανοποίησης Περιορισμών;

Το MapReduce είναι ένα γενικότερο πλαίσιο (framework) για την κατανομή και μαζική επεξεργασία πολλών δεδομένων (big data) σε έναν αυθαίρετα μεγάλο αριθμό υπολογιστών. Χρησιμοποιήθηκε αρχικά για την ευρετηριοποίηση ολόκληρου του Διαδικτύου. Με απλά λόγια δηλαδή, το MapReduce είναι η βάση της μηχανής αναζήτησης Google. Από εκεί και πέρα, το πλαίσιο MapReduce έχει χρησιμοποιηθεί και σε πλείστες άλλες εφαρμογές όπως η αυτόματη ομαδοποίηση εγγράφων, η μηχανική μάθηση και η αυτόματη μετάφραση.

Μία από τις καινοτομίες της διατριβής αφορά στην εφαρμογή του MapReduce πάνω στον Προγραμματισμό με Περιορισμούς. Πιο συγκεκριμένα, το σύνολο των υποψηφίων λύσεων ενός προβλήματος κωδικοποιείται και κατακερματίζεται σε

κομμάτια. Στη συνέχεια τα κομμάτια αυτά αποστέλλονται στους επιλυτές-εργάτες (mappers) για να αναζητήσουν αν μέσα σε αυτά κρύβεται κάποια πραγματική λύση του προβλήματος. Με άλλα λόγια, διαιρείται το δένδρο αναζήτησης σε πολλά κομμάτια τα οποία κατανέμονται αυτόματα μέσω του MapReduce σε έναν αριθμό εργατών που τα εξερευνούν ταυτόχρονα.

Επειδή το MapReduce δέχεται ως είσοδο μόνο αρχεία κειμένου, στη διατριβή προτείνεται ένας γρήγορος τρόπος κωδικοποίησης των κομματιών του δένδρου αναζήτησης και η αποθήκευσή τους σε ένα μεγάλο αρχείο κειμένου. Ο στόχος είναι να κομματιάσουμε το δένδρο σε όσο το δυνατόν πιο ισομεγέθη μέρη, ούτως ώστε να πετύχουμε δίκαιη κατανομή του φόρτου εργασίας στους επιλυτές-mappers.

Είναι ασύμφορο να διατρέξουμε όλο το δένδρο αναζήτησης προκειμένου να δούμε τη δομή και τη μορφή του και να το κομματιάσουμε σε ίσα μέρη. Γι' αυτό το λόγο, επισκεπτόμαστε δειγματοληπτικά μόνο κάποιους από τους κόμβους του δένδρου, για να σκιαγραφηθεί και να υπολογιστεί η δομή του, δίχως να χρειαστεί να τα επισκεφθούμε όλους τους κόμβους του εκ των προτέρων [P4].

3 Τα πλεονεκτήματα της χαλαρής διάδοσης περιορισμών

Μέχρι τώρα εστίασαμε στις μεθόδους αναζήτησης λύσεων για τα Προβλήματα Ικανοποίησης Περιορισμών και το πώς μπορούν να επιταχυνθούν και να κατανεμηθούν. Στα πραγματικά συστήματα Προγραμματισμού με Περιορισμούς όμως, η κάθε μέθοδος αναζήτησης εναλλάσσεται με τη λεγόμενη *διάδοση περιορισμών* (constraint propagation). Οπότε, είναι εξίσου σημαντικό να ασχοληθεί κανείς και με αυτήν.

3.1 Μέθοδοι αναζήτησης

Μία συστηματική μέθοδος αναζήτησης (search method) επιλύει βήμα βήμα ένα Πρόβλημα Ικανοποίησης Περιορισμών. Αρχικά, αναθέτει μία τιμή στην πρώτη μεταβλητή του προβλήματος από το πεδίο τιμών της. Έπειτα, αναθέτει στη δεύτερη μεταβλητή του προβλήματος μία τιμή από το πεδίο τιμών της. Μετά ανατίθεται τιμή στην τρίτη μεταβλητή κ.ο.κ.

Αν μετά από κάποια ανάθεση παραβιάζεται οποιοσδήποτε περιορισμός, δεν έχει νόημα να διατηρήσουμε αυτή την ανάθεση και να προχωρήσουμε, άσκοπα, στην επόμενη ανάθεση. Αυτό που πρέπει να κάνουμε σε αυτή την περίπτωση, είναι να αναιρέσουμε την «προβληματική» ανάθεση και να δοκιμάσουμε κάποια εναλλακτική.

3.2 Διάδοση περιορισμών

Η διάδοση περιορισμών εξυπηρετεί το να γλιτώνουμε άσκοπες αναθέσεις τιμών σε μεταβλητές. Αυτό δεν επιτυγχάνεται με το να ελέγχουμε απλά (όπως στις κλασικές μεθόδους αναζήτησης) αν οι υπάρχουσες αναθέσεις παραβιάζουν κάποιον περιορισμό. Η διάδοση περιορισμών αποσκοπεί επιπλέον στο να απομακρύνει από τα πεδία τιμών των υπολοίπων μεταβλητών (στις οποίες δεν έχει ανατεθεί ακόμα τιμή) τιμές οι οποίες είναι ασυνεπείς.

Με άλλα λόγια, η διάδοση περιορισμών δεν εστιάζει μόνο στις υπάρχουσες αναθέσεις, αλλά προσπαθεί να αφαιρέσει από τα πεδία τιμών όσες περισσότερες τιμές δεν μπορούν να συμμετέχουν σε μελλοντικές αναθέσεις. Σε γενικές γραμμές, όσες περισσότερες ασυνεπείς (inconsistent – no good) τιμές αφαιρούνται από τα πεδία τιμών, τόσο υψηλότερο επίπεδο διάδοσης περιορισμών λέμε ότι έχουμε.

3.3 Πόση διάδοση περιορισμών;

Εύλογα θα μπορούσε κανείς να ισχυριστεί λοιπόν ότι όσο περισσότερη διάδοση περιορισμών έχουμε, τόσο περισσότερο θα βοηθηθούν οι μέθοδοι αναζήτησης και θα κάνουν λιγότερες άσκοπες αναθέσεις τιμών. Αυξάνοντας όμως το επίπεδο διάδοσης περιορισμών, αυξάνεται αναπόφευκτα και ο χρόνος που αυτή δαπανά.

Οι σχετικές εργασίες στην τρέχουσα βιβλιογραφία εστιάζουν στην επινόηση υψηλών επιπέδων διάδοσης περιορισμών (higher-level consistencies). Το ζητούμενο όμως δεν είναι να μετακυλήσουμε το κόστος της μεθόδου αναζήτησης στη διάδοση περιορισμών, αλλά να μειώσουμε το χρόνο που απαιτείται αθροιστικά για τη λύση ενός Προβλήματος Ικανοποίησης Περιορισμών.

3.4 Από τη θεωρία στην πράξη

Αυτό που είναι άξιο απορίας –και αποτελεί αντικείμενο της έρευνας αυτής της διατριβής– είναι γιατί ενώ υπάρχουν πάρα πολλές σημαντικές εργασίες σχετικές με τα υψηλά επίπεδα διάδοσης περιορισμών, στις ίδιες εργασίες αυτές πλέον ομολογείται ότι σπάνια χρησιμοποιούνται στην πράξη. Εν τέλει, ποια θα μπορούσε να είναι μια πρακτική μορφή διάδοσης περιορισμών;

Μια απλή αλλά όχι απλοϊκή απάντηση είναι ότι πρακτική μορφή διάδοσης περιορισμών είναι αυτή που χρησιμοποιείται σε πρακτικά συστήματα Προγραμματισμού με Περιορισμούς. Με τη σειρά του, ένα πρακτικό σύστημα Προγραμματισμού με Περιορισμούς είναι εκείνο που

1. παρέχει ευκολία διατύπωσης προβλημάτων και έτσι χρησιμοποιείται από έναν ικανό αριθμό προγραμματιστών-χρηστών και
2. δύναται να επιλύσει σε ικανοποιητικούς χρόνους ένα ευρύ φάσμα Προβλημάτων Ικανοποίησης Περιορισμών: από απλές σπαζοκεφαλίες όπως η τοποθέτηση οκτώ βασιλισσών σε μια σκακίερα χωρίς να απειλούνται μεταξύ τους, μέχρι την κατάστρωση του ωρολογίου προγράμματος του Τμήματος Πληροφορικής και Τηλεπικοινωνιών.

Ο Naxos Solver είναι μια τέτοια βιβλιοθήκη Προγραμματισμού με Περιορισμούς και χρησιμοποιήθηκε ως το πεδίο εφαρμογής των πειραμάτων της διατριβής.

Για να κρίνουμε αν ένα επίπεδο διάδοσης περιορισμών είναι πρακτικό, χρειάζεται, εκτός από το να βρούμε ένα πρακτικό σύστημα Προγραμματισμού με Περιορισμούς, να αποφασίσουμε ποια Προβλήματα Ικανοποίησης Περιορισμών θα επιλύσουμε. Σε αυτή τη διατριβή, για να υπάρξει όσο το δυνατόν περισσότερη αμεροληψία, χρησιμοποιήθηκε ένα ευρύ φάσμα προβλημάτων, από τον πραγματικό κόσμο αλλά και τεχνητών, από τον πρώτο διεθνή διαγωνισμό μικρών επιλυτών

XCSP3.² Εξάλλου, οι περισσότερες εργασίες σε αυτή την ερευνητική περιοχή χρησιμοποιούν μέρος αυτών των προβλημάτων για πειραματικές μετρήσεις.

3.5 Τα πλεονεκτήματα της χαλαρής διάδοσης περιορισμών

Υπό το πρίσμα του γενικευμένου πρακτικού πειραματικού πλαισίου αυτής της εργασίας καταγράφηκαν εκτενείς παρατηρήσεις αναφορικά με την απόδοση διαφόρων επιπέδων διάδοσης περιορισμών, όπως η συνέπεια ακμών (arc consistency), η συνέπεια ορίων (bounds consistency) και μία καινούργια μορφή συνέπειας ορίων για μεταβλητές με μέγεθος πεδίου τιμών μικρότερο από k (k bounds consistency) που προτείνεται στη διατριβή [P5].

Γίνεται για πρώτη φορά καταγραφή των περιπτώσεων στις οποίες η διατήρηση της συνέπειας ορίων (ενός χαλαρού επιπέδου διάδοσης περιορισμών) κάνει γρηγορότερη την αναζήτηση από ό,τι η διατήρηση συνέπειας ακμών (ενός υψηλότερου επιπέδου διάδοσης περιορισμών). Αυτή η παρατήρηση από μόνη της είναι σημαντική, επειδή υπάρχει η κοινή πεποίθηση ότι τα υψηλότερα επίπεδα διάδοσης περιορισμών είναι πάντοτε πιο αποτελεσματικά.

Πέρα από την καταγραφή εκτενών παρατηρήσεων, η διατριβή επεξηγεί θεωρητικά το λόγο που κάποια Προβλήματα Ικανοποίησης Περιορισμών λύνονται αποδοτικότερα μέσω της διατήρησης συνέπειας ορίων σε σχέση με τη διατήρηση συνέπειας ακμών. Παρουσιάζονται οι επιμέρους υπολογιστικές πολυπλοκότητες και προτείνεται ένα κριτήριο για να επιλέγουμε το καλύτερο επίπεδο διάδοσης περιορισμών για το κάθε Πρόβλημα Ικανοποίησης Περιορισμών πριν αρχίσουμε να το επιλύουμε [P1].

Επίλογος

«Ο χρήστης απλά διατυπώνει το πρόβλημα και ο επιλυτής βρίσκει τη λύση.» Αυτό είναι το σύνθημα του επιστημονικού τομέα του Προγραμματισμού με Περιορισμούς και αυτόν τον στόχο επιχειρεί να θεραπεύσει η εν λόγω διατριβή: την καλύτερη εμπειρία του χρήστη μέσω της γρηγορότερης επίλυσης καθημερινών Προβλημάτων Ικανοποίησης Περιορισμών. Η συνεισφορά της διατριβής συνοψίζεται

1. στον συνδυασμό τυχαίων και ντετερμινιστικών ευρετικών κανόνων για την επιτάχυνση της αναζήτησης λύσεων [P2,P3],
2. στην κατανομή του δένδρου αναζήτησης σε πολλούς υπολογιστές-εργάτες με την αρχιτεκτονική MapReduce [P4], καθώς και
3. στην πειραματική αλλά και θεωρητική τεκμηρίωση των πλεονεκτημάτων που δύναται να προσφέρει η χαλάρωση της διάδοσης περιορισμών [P1,P5].

²<http://www.cril.univ-artois.fr/XCSP17>

CONTENTS

1	INTRODUCTION	31
1.1	What is Constraint Programming?	31
1.2	How Constraint Programming relates to AI?	32
1.3	How Constraint Programming relates to programming?	33
1.4	Our contributions	33
2	CONSTRAINT SATISFACTION PRELIMINARIES	35
2.1	Constraint Satisfaction Problems	35
2.1.1	Variants	37
2.1.2	Constraint networks	38
2.1.3	Map-coloring problem	38
2.1.4	Constrained optimization	39
2.2	A goal-driven search methods framework	39
2.2.1	Search methods are made up of goals	40
2.2.2	The Depth-First Search example	40
2.2.3	Defining DFS using goals	41
2.2.4	Example: Applying DFS on a CSP	41
2.2.5	Defining Iterative Broadening using goals	42
2.2.6	Search tree exploration	43
2.3	Constraint propagation	44
2.3.1	Example	44
2.3.2	Node consistency	45
2.3.3	Arc consistency	45
2.3.4	Path consistency	47
2.3.5	k -consistency	47
2.3.6	Interchanging constraint propagation with a search method	49
2.4	Naxos Solver: Our guinea pig	49
2.4.1	Constraints from a C++ programmer's perspective	50
2.4.2	Search methods as C++ classes	54
3	RELATED WORK	57
3.1	Heuristics exploitation in related work	57
3.1.1	Variable ordering heuristics	57
3.1.2	Value ordering heuristics	59
3.1.3	Heuristics in deterministic search methods	59
3.1.4	Heuristics in random search methods	60
3.1.5	Local search methods	60
3.1.6	Heuristics and probabilities	60

3.2	Distributing Constraint Programming with MapReduce	62
3.2.1	Mappers and reducers	62
3.2.2	Applications	63
3.2.3	In the battle against the pandemic	64
3.2.4	OR-parallelism vs. CSP partitioning	64
3.2.5	A MapReduce and CP combination and other related work	65
3.2.6	Partitioning the search space	65
3.3	Constraint propagation related work	66
3.3.1	Learning from mistakes or preventing them?	66
3.3.2	The importance of arc consistency	67
3.3.3	Higher-level consistencies (HLCs)	67
3.3.4	Toward more relaxed consistencies	69
3.3.5	Constraint propagation, validation, and explanation	69
4	BRIDGING THE GAP: FROM RANDOM TO DETERMINISTIC HEURIS-	
	TICS	71
4.1	New probabilistic heuristics	71
4.1.1	Heuristic estimation as a real number	71
4.1.2	Heuristics probabilistic foundations	72
4.1.3	Bridging the two opposites	74
4.1.4	Balanced heuristic distribution for two or more maximum heuristic values	76
4.2	Piece of Pie Search	77
4.2.1	The algorithm's core	77
4.2.2	Heuristic confidence vs. node level	81
4.2.3	POPSAMPLE average complexity	82
4.2.4	The motivation behind PoPS	82
4.3	Empirical evaluations	84
4.3.1	University course scheduling	84
4.3.2	Radio link frequency assignment	86
4.3.3	POPSAMPLE during hard optimization	86
4.3.4	PoPS vs. other search methods	88
4.4	Conclusions	88
5	CONSTRAINT PROGRAMMING MAPREDUCE'D	91
5.1	Optimal search tree partitioning	91
5.2	Encoding a search tree node into an array of integers	93
5.3	Search tree nodes random sampling	97
5.3.1	Pre-estimating a node's exploration time	97
5.3.2	Pre-estimating a node's descendants number	98
5.4	The MapReduce input specification	100
5.5	Slicing the search tree	100
5.5.1	Slicing the search tree by repeating sequential search	101
5.5.2	Slicing the search tree by mocking sequential search	103
5.5.3	How much does simulation cost?	107
5.5.4	Multiple MapReduce rounds	108
5.6	Empirical results with MapExplore	108
5.6.1	Sequential vs. simulation time	108

5.6.2	MapExplore parallel/distributed execution time	109
5.7	Conclusions	112
6	THE REVENGE OF BOUNDS CONSISTENCY	117
6.1	Consistency enforcement	117
6.2	Our contribution and alternative approaches	120
6.3	Constructive search	121
6.3.1	The typical backtracking search method	121
6.3.2	A search tree path	123
6.3.3	Paths vs. trees	123
6.4	Maintaining consistency during search	123
6.4.1	Time complexity in a search tree node	123
6.4.2	The constraint propagation aggregate complexity	125
6.4.3	Backup and restore aggregate complexity	126
6.4.4	Will arc or bounds consistency be faster?	129
6.4.5	Discussion	130
6.5	Empirical evaluations	131
6.5.1	Methodology	131
6.5.2	Execution	135
6.5.3	Visualization	136
6.5.4	Observations	136
6.6	The new k -bounds-consistency variant	137
6.6.1	Theoretical analysis	137
6.6.2	Empirical results	138
7	CONCLUSIONS AND FUTURE DIRECTIONS	141
7.1	Unified random and deterministic heuristics	141
7.2	Distributed Constraint Programming via MapReduce	141
7.3	Relaxed constraint propagation: Less is more	142
7.3.1	Predicting the efficiency of relaxed consistency	142
7.3.2	A new relaxed consistency variant	143
7.3.3	Toward one unified benchmarking	143
	REFERENCES	145

1. INTRODUCTION

None are more hopelessly enslaved than those who falsely believe they are free.

Johann Wolfgang von Goethe

With this quote, Goethe implies that we are all under strict *constraints*; they are an integral part of our lives, even when we do not admit it.

Therefore, a good approach to tackle any problem is to explicitly describe its constraints and search for a solution that does not violate them.

1.1 What is Constraint Programming?

Here Constraint Programming comes into the picture. Its motto is “the user simply states the problem and the computer solves it” [34]. This proposition implies that

- the “user” is required to provide only a bare minimum of the description of a problem, i.e. only the *constraints* should be defined, and
- the rest (solution search process) is undertaken by the machine, i.e. solver.

There are several variations of the solution search processes under the Constraint Programming umbrella. In all cases, however, we describe and formalize every problem as a *Constraint Satisfaction Problem (CSP)*. A formal description of a CSP will follow in the next chapter.

Conclusively, Constraint Programming is the set of all the methodologies that can solve arbitrary Constraint Satisfaction Problems (CSPs).

The description of any CSP is a minimal definition of *what* the problem is and does *not* contain information on *how* to solve it. Normally, a CSP has a very large number of candidate solutions, and a Constraint Programming methodology should be able to identify the feasible solutions out of them.

Constraint Programming allows the easy and declarative statement of a CSP and provides an “armory” of several generic search methods that can be used to solve it. Constraint Programming has been applied in scheduling [58], radio frequency assignment [19], Bioinformatics problems [6, 69], etc.



Figure 1.1: Harold Cohen, *040501*, print

1.2 How Constraint Programming relates to AI?

Artificial Intelligence (AI) is a prestigious Computer Science area that changes the world. Distinguished AI applications include self-driving cars, search engines, medical diagnosis, image recognition, even automatic drawing of paintings such as the ones illustrated in Fig. 1 and 1.1.

Computer vision is a discipline that traditionally belongs to AI. In 1975, David Waltz introduced constraint propagation, the core of Constraint Programming, to create a three-dimensional view of an object given a two-dimensional image [92]. Two years later, Alan Mackworth published in the journal of *Artificial Intelligence* the evolution of this constraint propagation algorithm which is with variations still the heart of most Constraint Programming solvers [55].

Constraint Programming also adopts search methods used in AI to solve Constraint Satisfaction Problems. A series of AI methods like depth-first search (DFS), limited discrepancy search (LDS), etc. have been successfully employed in the Constraint Programming world, during the CSP solving phase. Furthermore, Machine Learning (ML) is constantly gaining ground in the context of Constraint Programming [66].

AI is only one of the areas that have contributed to Constraint Programming so far. Integer Programming and Linear Programming that belong to Operations Research, another Computer Science discipline, have also influenced Constraint Programming.

1.3 How Constraint Programming relates to programming?

When one hears the term “Constraint Programming” for the first time, they often imagine that it is something like a common programming language. Nevertheless, the word “programming” here has a broader meaning.

Historically, Constraint Programming was implemented for the first time using Logic Programming. Thus, only the “Constraint Logic Programming” term initially existed.

When different implementations (imperative languages) came into the picture, the generalized term “Constraint Programming” was introduced.

Constraint Programming should not be mistaken for a tool to describe the steps toward solving a problem. Constraint Programming is all about describing the constraints that a solution should satisfy. Searching for a solution is done behind the scenes.

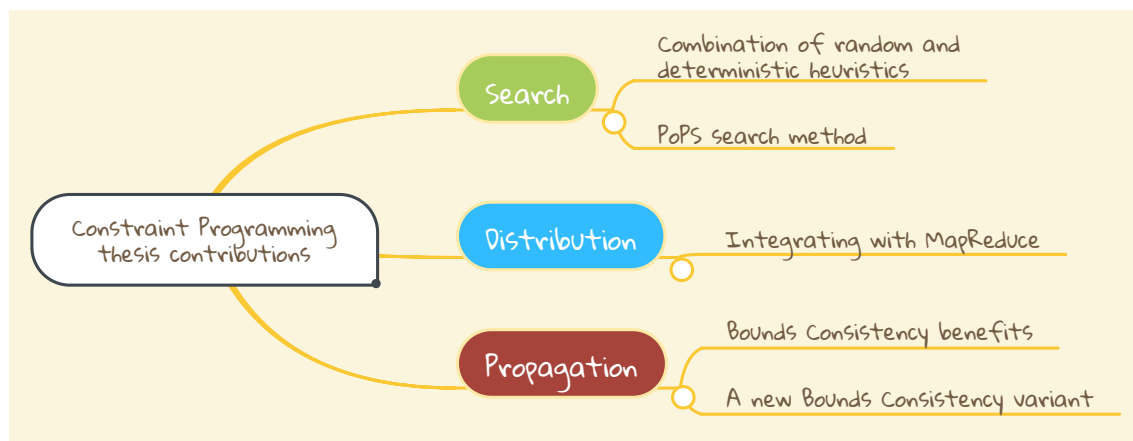


Figure 1.2: Our contributions

1.4 Our contributions

In programming languages, the statement of an algorithm is followed by compilation/interpretation and execution. In Constraint Programming, the statement of the constraints is followed by independent methodologies that solve the problem. This dissertation aims to make the solving process more efficient. Figure 1.2 summarizes our contributions in this direction.

The next Chapter 2 contains the preliminaries needed to understand our contributions in the rest of the chapters. Constraint Satisfaction Problems are formally defined along with a framework of search methods that solve them by employing heuristics. Constraint propagation, a basic element of Constraint Satisfaction, is introduced.

Chapter 3 goes through the papers that are relevant to our contributions and presents MapReduce.

Chapter 4 illustrates our first contribution [71, 72]. We study two distinct categories of heuristics, deterministic and random, and we make the most out of

them by smoothly combining them. We create new hybrid heuristics and a new search method based on them.

In Chapter 5, we integrate Constraint Programming into the state-of-the-art distributed MapReduce framework [70]. We exploit MapReduce scalability to solve large CSP instances.

Finally, in Chapter 6, we highlight the advantages of relaxed constraint propagation methodologies when used in conjunction with search methods to validate constraints and speed up search [73]. This last contribution attempts to contradict the “conventional wisdom” which implies that arc consistency or even higher consistency levels are always better than bounds consistency. We propose a new relaxed consistency level and a criterion to decide a priori when to enforce relaxed consistency instead of higher consistency levels [74].

2. CONSTRAINT SATISFACTION PRELIMINARIES

You see this risk time and again in deeply theoretical communities where they just solve problems for their own amusement and pretend that what they are doing has some utility. I should say, obviously, that I have been as guilty as the next person of doing this.

Jeffrey Ullman

Hopefully, Prof. Ullman was not thinking of Constraint Satisfaction Problems when stating the above; at least the nonartificial ones...

2.1 Constraint Satisfaction Problems

Constraint Programming (CP) aims at solving *Constraint Satisfaction Problems* (CSPs) in a transparent way: the user simply states the problem and the computer solves it [34]. The consequence of this “motto” is that the solver should decide automatically on its own which algorithm will solve a given CSP without human intervention; the role of the user is limited just to *define* the CSP.

This elegant separation of the user experience and the internal solving process is what makes Constraint Programming an intelligent paradigm, and this is the motivation behind this work. Every single CSP can be stated using commonplace formalizations [83, 88].

Definition 1. A *Constraint Satisfaction Problem* (CSP) consists of

- a set of constrained *variables* $\mathcal{X} = \{X_1, X_2, \dots, X_n\}$,
- the corresponding set of *domains* $\mathcal{D} = \{D_1, D_2, \dots, D_n\}$ which are finite sets (of integer values in this work) and
- the set of *constraints* between the variables $\mathcal{C} = \{C_1, C_2, \dots, C_e\}$. Each constraint is defined as $C_i = (S_i, R_i)$.
 - S_i is the subset of \mathcal{X} containing the variables affected by the constraint.
 - R_i contains all the valid combinations of the values of the domains of the variables in S_i . Formally, $R_i \subseteq D_{i_1} \times D_{i_2} \times \dots \times D_{i_k}$, with $X_{i_1}, X_{i_2}, \dots, X_{i_k} \in S_i$ and $i_1 < i_2 < \dots < i_k$.

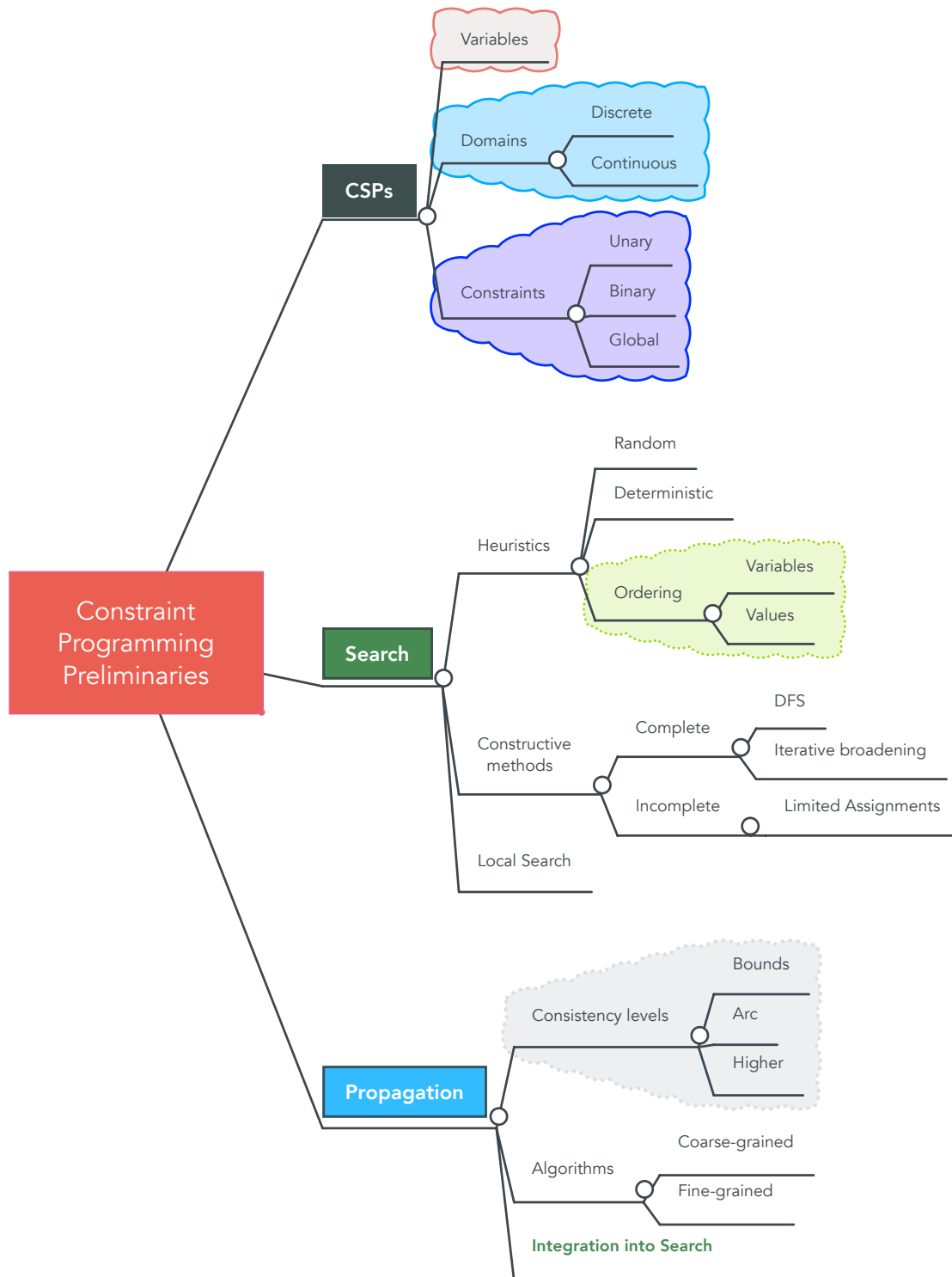


Figure 2.1: A “mind map” with Constraint Programming preliminaries

For the sake of readability, in this work, D_X also denotes the domain of the constrained variable X . Therefore, the domain of X_1 is denoted as D_1 and as D_{X_1} too; these two are equivalent.

If we assign a value to every variable, and the assignments are valid with respect to every constraint, then the assignment is a *solution*.

2.1.1 Variants

The above definition is very generic, thus many CSP categories-variants have been introduced so far depending on their type of domains and constraints.

Continuous vs. finite domains

There are CSPs where the domains are continuous sets of real numbers. For example, a variable can be assigned a decimal number, e.g. 0.25 or 0.7, out of $[0, 1]$. Nevertheless, in this work we study only CSPs that are described using finite sets of integers.

Unary constraints

We do not also refer to unary constraints in this work, due to their triviality. A unary constraint applies onto a single variable. The initial domain of each variable should be shrunk to include only the values permitted by the corresponding unary constraint. This action can be performed as a single preprocessing step, before proceeding to actually solve a CSP.

Binary constraints

Each binary constraint of a CSP affects exactly *two* variables. A binary CSP is a CSP containing only binary constraints. Binary CSPs are important because it has been proven that any non-binary CSP can be transformed into an equivalent binary one [81]. And it is easier, at least in theory, to manipulate a binary constraint rather than a higher-level constraint.

Let us assume that for a given binary CSP there is a unique constraint $C_k = (S_k, R_k)$ with $S_k = \{X_i, X_j\}$. Then, for the sake of readability, we denote this C_k constraint as C_{ij} , which in turn is equivalent to C_{ji} .

n -ary constraints

If a constraint affects n variables, we call it an n -ary constraint. The term “ n -ary” with $n > 2$ is usually used to differentiate a constraint from the unary and binary constraints.

In the general case, an n -ary constraint C_k refers to a fixed number of variables $n = |S_k|$ that can be assigned values out of an arbitrary set of tuples R_k .

With the more specialized “global constraint” term we refer to a constraint pattern such as “all the variables in S_k should be assigned different values” that can be applied to any number n of constraints variables.

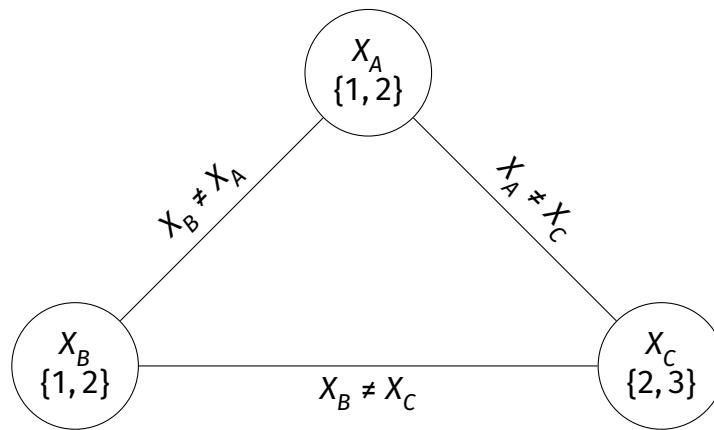


Figure 2.2: A constraint network

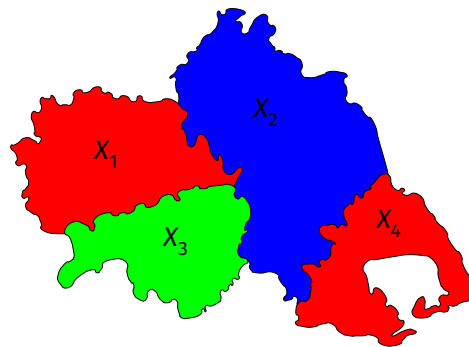


Figure 2.3: The four Thessaly prefectures

2.1.2 Constraint networks

It is easy to view a binary CSP as a graph. We consider its variables as the *nodes* of the graph, and we consider the constraints between the variables as the *edges* or *arcs* that connect the nodes (Fig. 2.2).

Even when Ugo Montanari first stated the CSP definition back in 1974, it was made clear that the notions of a constraint network and a binary CSP are interchangeable. Non-binary CSPs can be depicted as hypergraphs [61].

2.1.3 Map-coloring problem

CSPs cover a wide range of problems, including planning and scheduling [9], logic puzzles [47], all Boolean satisfiability problems [65], circuit design [76], robotics [56], and many others. CSPs are widespread because they express many problems that occur in real life.

There exists a huge list of interesting CSPs [38, 39]. For example, *map-coloring* is a CSP for assigning colors to each prefecture in a given map, so as no neighboring prefectures have the same color. Figure 2.3 illustrates a map of the Greek region “Thessaly,” containing four prefectures; the colors in the figure form an indicative solution.

Problem 1. Typically, “*Thessaly-coloring*” is a CSP with:

1. Four constrained variables: X_1, X_2, X_3, X_4 . Each one of them represents a prefecture color.
2. The corresponding domains are $D_{X_1} = D_{X_3} = \{1, 2\}$ and $D_{X_2} = D_{X_4} = \{1, 3\}$. Numbers **1 2 3** represent respectively red, green, blue.¹
3. The constraints are $X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3$, and $X_2 \neq X_4$.

The solution in Fig. 2.3 is represented by the assignment

$$\{X_1 \leftarrow \mathbf{1}, X_2 \leftarrow \mathbf{3}, X_3 \leftarrow \mathbf{2}, X_4 \leftarrow \mathbf{1}\}. \quad (2.1)$$

2.1.4 Constrained optimization

A variation of Constraint Satisfaction Problems (CSPs) is the so-called *Constrained Optimization Problems* (COPs). A COP consists of variables, domains, and constraints, just like any CSP. The difference is that a COP also requires an *objective function* which maps any solution to a number, which is called the *cost* of the solution. The target while solving a COP is not just to find a solution, but to find a *best* solution, i.e. a solution with a minimum cost.²

COPs can be solved like CSPs, using a *branch and bound* strategy: When a solution is found, its cost is recorded, and a new constraint is added to guarantee that the next solution will have a smaller cost than the recorded one.

In relation to CSP solving, the only additional requirement of the above COP solving procedure is adding dynamically a new constraint while searching. This makes it compatible with plain CSP search methods, so this work covers both CSPs and COPs as a whole.

On the other hand, this work does not cover *convex optimization*, a variant introduced in Mathematics which paved the way for advances in Computer Science [16]. Besides, convex optimization applies to *continuous* domains, e.g. $[0.5, 3.1]$, while in Constraint Programming we focus on *discrete* domains of constrained variables, e.g. $\{1, 2, 3\}$.

2.2 A goal-driven search methods framework

Apart from a way to state CSPs, a user/programmer needs an elegant way to state search methods that solve them. The CSPs should be “search-methods-agnostic,” while the search methods should be “CSP-agnostic” in order to keep the independence between Constraint Programming stages.

In related works, a lot of search methods have been implemented “out of the box” in modern solvers [36]. This means, at least to our knowledge, that the implemented search methods are coupled with the existing solvers. Nevertheless, in this work we use Naxos, a Constraint Programming solver created by us that not only supports the definition of ad hoc CSPs, but also facilitates the definition of “custom” search methods [67].

¹We could initially set all the domains equal to $\{1, 2, 3\}$. We used smaller initial domains just to simplify the problem.

²There is also a COP variation which requires to find the solution with the *maximum* cost. However, for simplicity reasons, we will not focus on it, as it can be easily transformed into a COP with a minimization objective.

```

1: function DFS( $\ell$ )
    ▷ The method reached the search tree level  $\ell$ :
2:    $D'_\ell \leftarrow D_\ell$ 
3:   for each  $v \in D'_\ell$  do
4:      $D_\ell \leftarrow \{v\}$    ▷ Assign  $v$  to  $X_\ell$ 
5:     if no constraint is violated then
6:       ▷ Proceed to the next variable/level:
7:       if  $\ell = n$  then
8:         return success
9:       else if DFS( $\ell + 1$ ) = success then
10:        return success
11:      end if
12:    end for
13:     $D_\ell \leftarrow D'_\ell$ 
14:    return failure
15: end function

```

Figure 2.4: Defining DFS using an imperative pseudocode language

2.2.1 Search methods are made up of goals

Every constructive search method is built up of goals. Each goal executes an operation, e.g. an assignment of a value to a constrained variable, and/or returns another goal to be executed. The goal returned can be a *meta-goal*, which is a goal that refers to another two goals. There are two meta-goal kinds:

1. The AND(g_1, g_2), which implies that the two sub-goals g_1 and g_2 must be executed and succeed both.
2. The OR(g_1, g_2), which executes g_1 . If g_1 does not succeed, i.e. if it does not lead to a solution, then g_2 is executed.

This goal-driven framework is able to describe most of the common search methods.

Note that even if we do not explicitly state it for the sake of simplicity, after each goal is executed, we ensure that every constraint is respected. If any constraint is violated after a goal's execution, the goal is considered as failed.

2.2.2 The Depth-First Search example

Depth-first search (DFS) is an elementary search method also known as *backtracking* search in the Constraint Programming world. This method iterates through the variables of a CSP. For each variable X selected, it selects a value v from its domain and makes the assignment $X \leftarrow v$. It subsequently proceeds to the next unassigned variable and makes another assignment, etc. as in Fig. 2.4.

If every variable is assigned a value and no constraint is violated, the assignments set comprises a *solution*. In any case, if there is a constraint violated, the last assignment to a variable is undone and we try to assign another value from its

domain. If all the alternative values are exhausted, we *backtrack* to the previous variable selected and we undo its assignment and so forth.

2.2.3 Defining DFS using goals

DFS can be straightforwardly described via goals. The ultimate goal in DFS and in every constructive search method is to `Label` every variable with a value. Each `Label`'s call aims to `Instantiate` a variable.

- $\text{DFS}(\mathcal{X}) := \text{Label}(\mathcal{X})$.
- $\text{Label}(\emptyset) := \text{success}$.
- $\text{Label}(\mathcal{X}) := \text{AND}(\text{Instantiate}(X), \text{Label}(\mathcal{X} - \{X\}))$, with $X \in \mathcal{X}$,

where \mathcal{X} is the set of all the variables. While `Label` iterates recursively through the CSP variables, an `Instantiate` call attempts to assign a selected value v to the variable X . If the assignment fails to produce a solution, the value v is deleted and another instantiation is attempted, until all the alternatives in D_X are exhausted.

- $\text{Instantiate}(X) := \text{failure}$, with $D_X = \emptyset$.
- $\text{Instantiate}(X) := \text{OR}(X \leftarrow v, \text{AND}(D_X \leftarrow D_X - \{v\}, \text{Instantiate}(X)))$, with $v \in D_X$.

The interdependencies between the above DFS goals are graphically displayed in Fig. 2.5.

Again, please note that in the above DFS description, we have omitted to check the constraints. Nevertheless, as we have stated in the previous section, after each goal is executed, it is implied that we check that no constraint is violated. This check is very important when we make an assignment or when a domain is modified. If any constraint is violated, the goal is automatically marked as failed by the framework.

2.2.4 Example: Applying DFS on a CSP

Let us search for a solution of the “Thessaly-coloring” Problem 1 by applying DFS on it.

- We begin by adding the $\text{DFS}(\mathcal{X})$ goal which is substituted by $\text{Label}(\mathcal{X})$. This will generate the rest of the goals.
- According to the above DFS definition, $\text{Label}(\mathcal{X})$ will be substituted by $\text{AND}(\text{Instantiate}(X_1), \text{Label}(\{X_2, X_3, X_4\}))$.
 1. The first subgoal is $\text{Instantiate}(X_1)$ which in turn implies $\text{OR}(X_1 \leftarrow 1, \text{AND}(D_{X_1} \leftarrow D_{X_1} - \{1\}, \text{Instantiate}(X_1)))$.
This means that we assign the value **1** to X_1 . If this goal or the subsequent goals fail, we will revoke their changes and execute $\text{AND}(D_{X_1} \leftarrow D_{X_1} - \{1\}, \dots)$.

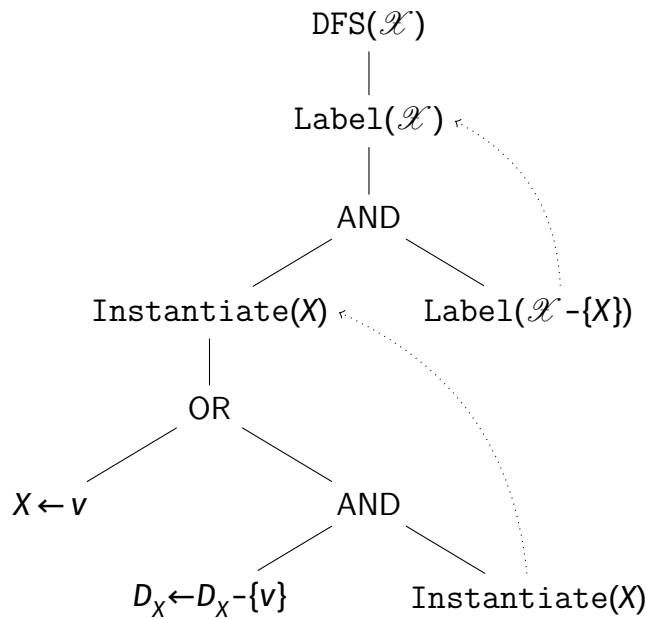


Figure 2.5: The combination of the goals that compose DFS

2. The second subgoal is $\text{Label}(\{X_2, X_3, X_4\})$. This will be substituted by $\text{AND}(\text{Instantiate}(X_2), \text{Label}(\{X_3, X_4\}))$.
 - $\text{Instantiate}(X_2)$ in turn implies $\text{OR}(X_2 \leftarrow 1, \text{AND}(D_{X_2} \leftarrow D_{X_2} - \{1\}, \text{Instantiate}(X_2)))$.
 - Nevertheless, the assignment $X_2 \leftarrow 1$ violates the constraint $X_1 \neq X_2$ and we will have to revert to $\text{AND}(D_{X_2} \leftarrow D_{X_2} - \{1\}, \dots)$. This will eventually generate $X_2 \leftarrow 2$.

And we continue to execute goals which in turn may generate other goals, until every one of them is satisfied.

2.2.5 Defining Iterative Broadening using goals

Figure 2.6 displays the corresponding goals' graph for the *Iterative Broadening* search method [42]. The goals' structure is similar to DFS. However, one basic difference is that there is one more level, namely *Broadening*, above the ordinary DFS goals.

- $\text{Broadening}(\mathcal{X}, \text{Breadth}) := \text{failure}$, if $\text{Breadth} > d$,
- $\text{Broadening}(\mathcal{X}, \text{Breadth}) := \text{OR}(\text{Label}(\mathcal{X}, \text{Breadth}), \text{Broadening}(\mathcal{X}, \text{Breadth} + 1))$, otherwise.

For each *Iterative Broadening* iteration, the *Breadth* parameter defines the maximum number of values that a constrained variable can be successively assigned. This value is initially 1. The *Breadth* value cannot exceed d , which in this context is the maximum cardinality (size) of the domains of all constrained variables. If *Breadth* exceeds d , *Broadening* fails.

Therefore, a second basic difference in comparison with DFS comes into play. The *Instantiate* goal takes now two more arguments.

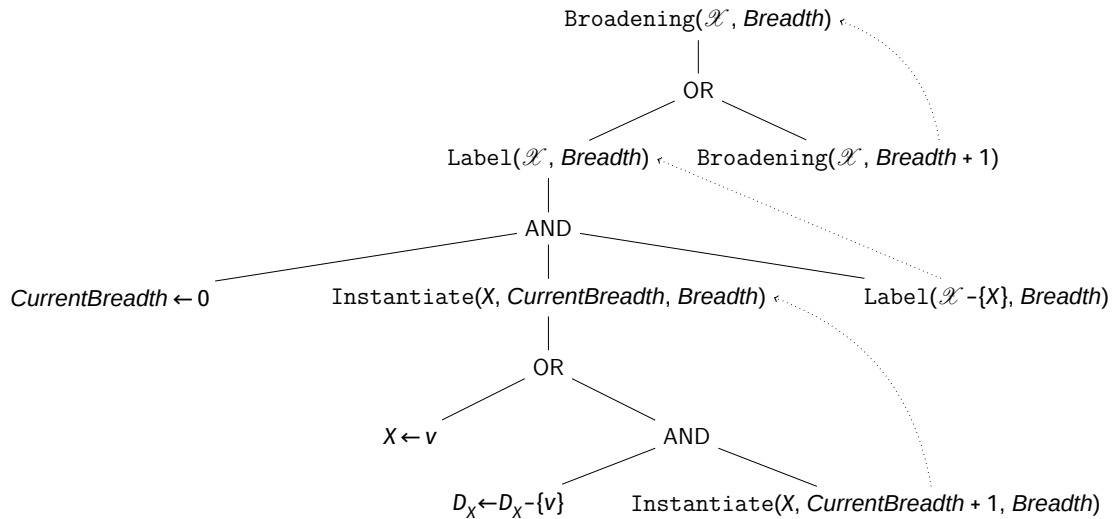


Figure 2.6: The goals composing Iterative Broadening

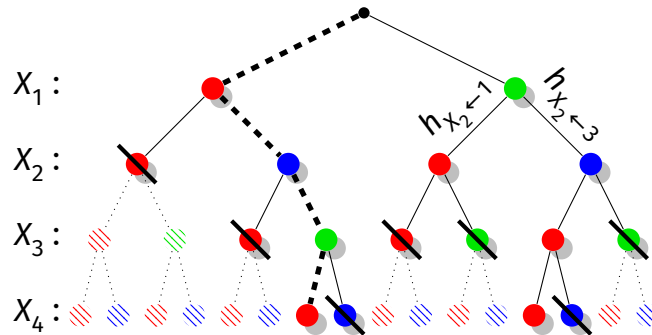


Figure 2.7: The search tree for Thessaly-coloring

- $\text{Instantiate}(X, \text{CurrentBreadth}, \text{Breadth}) := \text{failure}$, if $D_X = \emptyset$,
- $\text{Instantiate}(X, \text{CurrentBreadth}, \text{Breadth}) := \text{failure}$, if $\text{CurrentBreadth} > \text{Breadth}$,
- $\text{Instantiate}(X, \text{CurrentBreadth}, \text{Breadth}) := \text{OR}(X \leftarrow v, \text{AND}(D_X \leftarrow D_X - \{v\}, \text{Instantiate}(X, \text{CurrentBreadth} + 1, \text{Breadth})))$, otherwise.

This implements Iterative Broadening's semantics: The number of consecutive instantiations to the same variable cannot exceed *Breadth*.

2.2.6 Search tree exploration

A *search tree* is a descriptive way to depict every possible assignment in a CSP, such as map-coloring. Figure 2.7 displays the search tree for the Thessaly-coloring problem. The struck-out nodes have been pruned as nogoods.

Each path from the root (i.e. the uppermost node) represents an *assignment*. If the path from the root ends up into a leaf (lowest node), we have a *complete* assignment. E.g., the dotted path in Fig. 2.7 is an alternative form of the solution assignment in (2.1).

Each node is extended into two (or more) branches that represent two alternative choices. The presented search methods framework naturally supports distributed search methods. The left and right branches of some selected nodes can be explored concurrently to reduce the total tree exploration time. There are many different approaches regarding which nodes should be selected in order to split their two sub-trees [70, 79]. In Chapter 5, we distribute the search tree using the MapReduce approach.

2.3 Constraint propagation

Constraint propagation comes along with the aforementioned search methods and is interchanged with them. It is used to narrow the search trees by removing nogood values out of the domains of the constrained variables.

2.3.1 Example

Let us say that we have to crack a password $\langle X_1, X_2, X_3 \rangle$ consisting of three decimal digits. In order to guess them, we are given hints for five combinations.

1	$\langle 6, 8, 2 \rangle$	One number is correct and well placed
2	$\langle 6, 1, 4 \rangle$	One number is correct but wrongly placed
3	$\langle 2, 0, 6 \rangle$	Two numbers are correct but wrongly placed
4	$\langle 7, 3, 8 \rangle$	Nothing is correct
5	$\langle 7, 8, 0 \rangle$	One number is correct but wrongly placed

We can naturally model this puzzle as a constraint satisfaction problem of three variables X_1, X_2, X_3 , with initial domains $D_1 = D_2 = D_3 = \{0, 1, 2, \dots, 9\}$.

To solve this puzzle, it is *not* necessary to iterate through all the 1,000 candidate passwords and check them against the given constraints. An intelligent method would *propagate* the constraints of the above table.

- From the 4th constraint, we conclude that the domains of the variables do not contain 7, 3, and 8.
- From the above and the 5th constraint, we conclude that $X_1 = 0$ or $X_2 = 0$. After all, the values 7 and 8 are incorrect, so 0 is correct, but wrongly placed.
- From the above and the 3rd constraint, we conclude that $X_1 = 0$, as we are told that 0 is wrongly placed as the second digit.
- From the first two constraints, 6 cannot be a correct digit.
- Therefore, from the 1st constraint, $X_3 = 2$, as neither 6 nor 8 can be correct digits.
- Finally, from the 2nd constraint, $X_2 = 4$, because if 1 was correct, we should assign it either to X_1 or to X_3 , which would contradict the above.

This was a *constraint propagation* example that directly gave the solution $\langle X_1, X_2, X_3 \rangle = \langle 0, 4, 2 \rangle$. Normally, in other CSPs, constraint propagation should be combined with a search method. But in any case, it is apparent that propagation can dramatically reduce the search space, i.e. the set of candidate solutions that we should check.

2.3.2 Node consistency

Consistency is a very useful property in the road to solve a CSP. It implies that the values of the domains of each variable have a kind of *support* with respect to the CSP constraints.

The most trivial consistency form is *node consistency*, which implies that the domain of every variable should support the unary constraint where the variable is involved.

Example 1. Let X_1 and X_2 be two constrained variables with domains $D_1 = \{1, 2, 3\}$ and $D_2 = \{5, 6, 7, 8, 9\}$ and the respective unary constraints $C_1 = (\{X_1\}, \{(1), (2), (3)\})$ and $C_2 = (\{X_2\}, \{(5), (6), (7)\})$ which can be simply stated as $X_2 < 8$.

X_1 is node consistent as all its values are included in the respective unary constraint C_1 . On the other hand, X_2 is node inconsistent, as the values 8 and 9 in D_2 are not supported in C_2 .

2.3.3 Arc consistency

Definition 2. An arc (X_i, X_j) is *arc consistent* iff for each $v_i \in D_i$ there exists a $v_j \in D_j$ with (v_i, v_j) not violating C_{ij} .

Example 2. Let X_1 and X_2 be two constrained variables with domains $D_1 = \{1, 2, 3\}$ and $D_2 = \{5, 6, 7, 8, 9\}$. Let us assume that the constraint between the variables is $X_2 = 4 + X_1$.

(X_1, X_2) is arc consistent, as for each of the values 1, 2, 3 in D_1 , the corresponding values 5, 6, 7 belong to D_2 .

On the other hand, (X_2, X_1) is *not* arc consistent. To prove this, we need just one value from D_2 that does not have any support in D_1 . Indeed, for the value 8 in D_2 , there is not any v_1 in D_1 with $4 + v_1 = 8$.

If we want to make (X_2, X_1) arc consistent, we should remove the values 8 and 9 out of D_2 as they do not have any supports in D_1 .

This example also illustrates that consistency is not a symmetric property.

In order to check if an arc (X_i, X_j) is arc consistent, we have to iterate through all the values of D_i . The function that does this and removes the unsupported values from D_j is called REVISE.

```

function REVISE( $X_i, X_j$ )
    domain_is_modified  $\leftarrow$  false
    for each  $v_i \in D_i$  do
        value_is_supported  $\leftarrow$  false
        for each  $v_j \in D_j$  do
            if  $(v_i, v_j) \in R_{ij}$ , with  $C_{ij} \in \mathcal{C}$  then
    
```

```

        value_is_supported ← true
        break
    end if
end for
if value_is_supported then
    continue
else
    Remove  $v_i$  out of  $D_i$ 
    domain_is_modified ← true
end if
end for
return domain_is_modified
end function

```

REVISE returns true when it has removed at least one value out of a domain.

Coarse-grained vs. fine-grained arc consistency algorithms

REVISE function is able to make consistent just one arc. What about making the whole constraint network arc consistent? There is a whole family of arc consistency algorithms, and AC-3 is the most prominent and used one.

```

function AC-3
     $Q \leftarrow \{(X_i, X_j) \mid C_{ij} \in \mathcal{C}\}$ 
    while  $Q \neq \emptyset$  do
        Remove an arc  $(X_i, X_j)$  out of  $Q$ 
        if REVISE( $X_i, X_j$ ) then
             $Q \leftarrow Q \cup \{(X_k, X_j) \mid C_{ki} \in \mathcal{C}, k \neq j\}$ 
        end if
    end while
end function

```

As AC-3 initially puts every arc into the Q , all constraints are revised. When a domain of a variable X_i is modified by the REVISE function, the modification is *propagated* to the other constrained variables which are linked to X_i via an arc.

The algorithms, such as AC-3, that propagate the removal of a value out of a domain to the other linked constrained *variables* (using a queue of arcs) are called *coarse-grained* arc consistency enforcement algorithms.

On the other hand, the algorithms which propagate the removal of a value using a queue that apart from arcs also contains *values* of domains, are called *fine-grained* arc consistency enforcement algorithms.

Fine-grained algorithms are typically faster than coarse-grained ones. Nevertheless, they require complex data structures to propagate modifications to (a bigger set of) domain values rather than to (a smaller set of) variables as coarse-grained algorithms do, and they are unavoidably not used in practice. After all, Bessiere et al. eventually constructed a coarse-grained algorithm that performs as fast as its fine-grained counterparts [12].

Maintaining arc consistency

As illustrated in Fig. 2.8, arc consistency does not necessarily imply that we have a solution. Therefore, in order to create a solution, we have to combine arc consistency with a search method. After all, arc consistency reduces the search space that a search method—such as depth first search (DFS) or limited discrepancy search (LDS) etc.—has to explore.

According to the *maintaining arc consistency* (MAC) methodology, we should begin searching for a solution by repeating the assignment of values to variables and by checking every time—e.g. after each assignment—if the constraint network is arc consistent. If an assignment causes an inconsistency, then it should be canceled, and another value should be chosen.

2.3.4 Path consistency

Arc consistency each time involves only two variables and checks whether the values of the one variable are supported by the values of the other variable. Nevertheless, this has the drawback of focusing only on a part of the CSP and loses sight of the big picture.

Consider the CSP in Fig. 2.8a. All the arcs are consistent but the CSP itself is “inconsistent” as it has no solution. That is why this type of consistencies, such as arc consistency, are called *local*. Because they affect a subset of the variables or constraints and do not imply the complete CSP consistency.

Path consistency is another local consistency variant that is broader than arc consistency in the sense that it involves more than two variables.

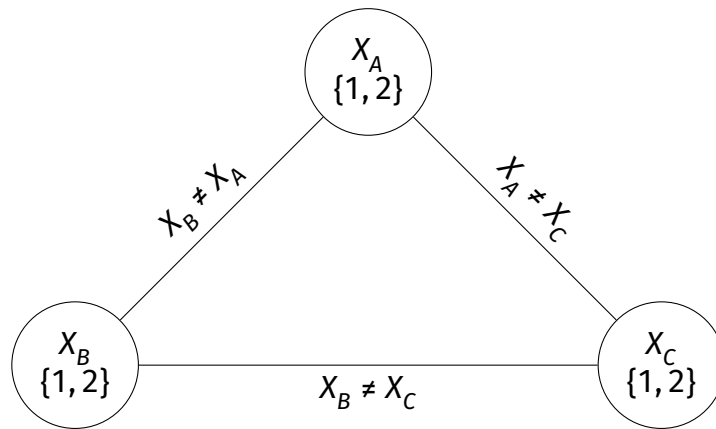
Definition 3. The variables X_{i_1} and X_{i_m} are *path consistent* if and only if for every pair of values $(v_{i_1}, v_{i_m}) \in D_{i_1} \times D_{i_m}$ that satisfies the $C_{i_1 i_m}$ constraint (and the respective unary constraints) and for every path of variables $(X_{i_1}, X_{i_2}, \dots, X_{i_m})$ there are values $(v_{i_2}, \dots, v_{i_{m-1}}) \in D_{i_2} \times \dots \times D_{i_{m-1}}$ so that $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots,$ and $(v_{i_{m-1}}, v_{i_m})$ satisfy the respective binary (and unary) constraints.

Evidently, the CSP in Fig. 2.8a is path *inconsistent*, as for each valid combination of the values of the first two variables, there does not exist a value in the third variable that supports it.

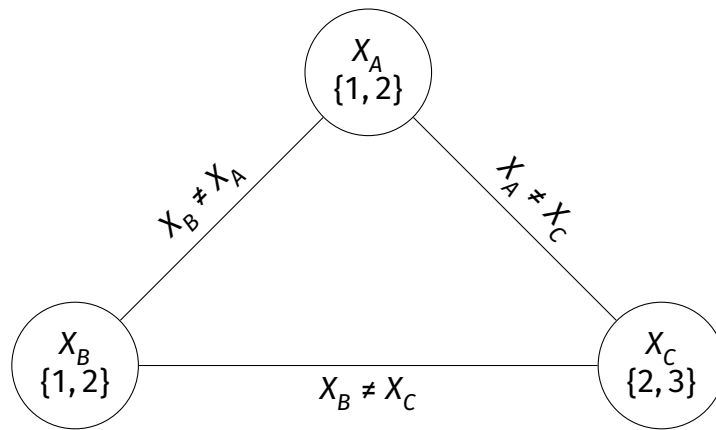
2.3.5 k -consistency

Node consistency involves one variable, arc consistency has to do with two variables, and path consistency with three. What about an arbitrary number of k variables?

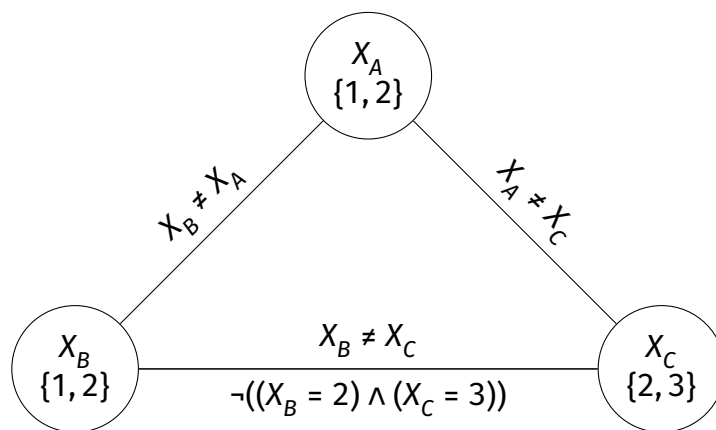
Definition 4. The variables $X_{i_1}, X_{i_2}, \dots, X_{i_{k-1}}$ are *k -consistent* with respect to the variable X_{i_k} if and only if for every tuple of values $(v_{i_1}, v_{i_2}, \dots, v_{i_{k-1}}) \in D_{i_1} \times D_{i_2} \times \dots \times D_{i_{k-1}}$ that satisfies the corresponding constraints (of the $k - 1$ variables) there is a value $v_{i_k} \in D_{i_k}$ so that the extended tuple $(v_{i_1}, v_{i_2}, \dots, v_{i_{k-1}}, v_{i_k})$ satisfies all the constraints between the k variables.



(a) There is no solution



(b) There are two solutions



(c) There is a unique solution

Figure 2.8: Three arc consistent constraint networks

1-consistency is identical to node consistency, 2-consistency coincides with arc consistency, and 3-consistency is equivalent to path consistency if the CSP is binary, as path consistency is only defined for binary CSPs, while 3-consistency may involve ternary constraint too.

2.3.6 Interchanging constraint propagation with a search method

Constraint propagation is a form of inference [10] and reasoning [11]. It is an intelligent methodology incorporated in constraint programming solvers. A common usage in such solvers is to execute constraint propagation after each search method step, i.e. after each single search method goal execution.

Following the example in section 2.2.4, let us search for a solution of the “Thessaly-coloring” Problem 1 by applying DFS on it. After each search method step, we propagate constraints trying to reduce the overall needed steps.

- We begin by adding the $\text{Label}(\mathcal{X})$ goal. This will generate the rest of the goals. Constraint propagation execution does not have now any effect.
- $\text{Label}(\mathcal{X})$ generates $\text{AND}(\text{Instantiate}(X_1), \text{Label}(\{X_2, X_3, X_4\}))$.
 1. The first subgoal is $\text{Instantiate}(X_1)$ which in turn implies $\text{OR}(X_1 \leftarrow 1, \text{AND}(D_{X_1} \leftarrow D_{X_1} - \{1\}, \text{Instantiate}(X_1)))$.
This means that we assign the value **1** to X_1 . Due to the $X_1 \neq X_2$ and $X_1 \neq X_3$ constraints, constraint propagation removes this value out of D_2 and D_3 .
 2. The second subgoal is $\text{Label}(\{X_2, X_3, X_4\})$. This will be substituted by $\text{AND}(\text{Instantiate}(X_2), \text{Label}(\{X_3, X_4\}))$.
 - $\text{Instantiate}(X_2)$ in turn implies $\text{OR}(X_2 \leftarrow 2, \text{AND}(D_{X_2} \leftarrow D_{X_2} - \{2\}, \text{Instantiate}(X_2)))$.
 - We continue to execute goals which in turn may generate other goals, until every one of them is satisfied.

Please note that in comparison with the example in section 2.2.4, we have already, even in the execution of the first goals, one unnecessary step ($X_2 \leftarrow 1$) truncated due to constraint propagation, as the nogood value **1** is proactively removed out of D_{X_2} .

2.4 Naxos Solver: Our guinea pig

More than fifteen years ago, we started to create the Constraint Programming NAXOS SOLVER that supports all the aforementioned classic Constraint Satisfaction properties. We published it as an open-source project and today it is used by a broad audience.³

To ensure that the contributions of this dissertation will be used in practice, we first applied and tested them in this real solver. Otherwise, our proposals would

³<https://github.com/pothitos/naxos>



be “like shooting an arrow into the air and, where it lands, painting a target” as the chemist Homer Burton Adkins (1892–1949) once said.

2.4.1 Constraints from a C++ programmer’s perspective

The N Queens problem

Definition. In the N queens problem, we should place N queens on an $N \times N$ chessboard, so that no queen is attacked. In other words, we should place N items on an $N \times N$ grid, in a way that no two items share the same line, column or diagonal. Figure 2.9 displays an example for $N = 8$. The eight queens are not attacked.

In each column $0, 1, \dots, N - 1$ we will have a queen. It remains to find out the *line* where each queen will be placed. Therefore, we ask to assign values to the variables X_i with $0 \leq X_i \leq N - 1$, where X_i is the line on which the queen of column i will be placed.

Regarding the constraints, first of all, no two queens should share the same line, i.e.

$$X_i \neq X_j, \forall i \neq j. \quad (2.2)$$

They should not also share the same diagonal, consequently

$$X_i + i \neq X_j + j \quad \text{and} \quad X_i - i \neq X_j - j, \forall i \neq j. \quad (2.3)$$

$X_i - i$ corresponds to the primary diagonal and $X_i + i$ to the secondary diagonal of the queen of column i .

Code. In the solver code, the variables X_i are represented by the array `Var`, that according to (2.2) should have different elements. Concerning (2.3), we create two other arrays, namely `VarPlus` and `VarMinus`, with the elements $X_i + i$ and $X_i - i$ respectively. For these arrays we will also declare that their elements shall be different between them. The relevant C++ code follows.

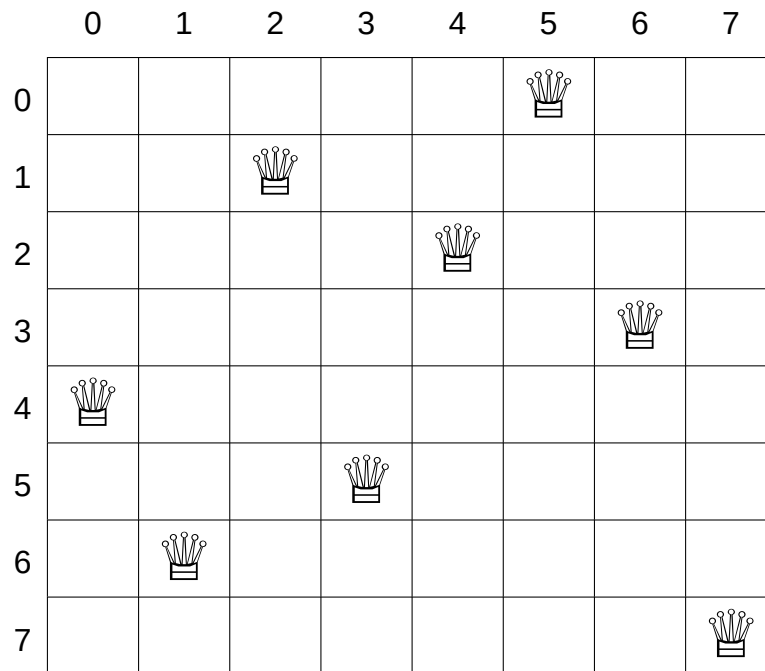


Figure 2.9: Eight queens not attacking each other

```

int N = 8;
NsProblemManager pm;

NsIntVarArray Var, VarPlus, VarMinus;
for (int i = 0; i < N; ++i) {
    Var.push_back(NsIntVar(pm, 0, N-1));
    VarPlus.push_back(Var[i] + i);
    VarMinus.push_back(Var[i] - i);
}
pm.add(NsAllDiff(Var));
pm.add(NsAllDiff(VarPlus));
pm.add(NsAllDiff(VarMinus));

pm.addGoal(new NsgLabeling(Var));
while (pm.nextSolution() != false)
    cout << "Solution: " << Var << "\n";

```

SEND + MORE = MONEY

Another example is a known *cryptarithm* problem.

Definition. In cryptarithms we have some arithmetic relations between words, such as *SEND + MORE = MONEY*. Each letter of the words represents a specific digit from 0 to 9; thus, each word represents a decimal number. Two different letters should not represent the same digit. E.g. for the equation *SEND + MORE = MONEY*, we will put the same digit in the positions where *E* appears. The same

applies for the rest of the letters, that should however be assigned different digits than the one for *E*. After all the assignments, the relation of the cryptarithm should be valid.

Code. The problem declaration for the NAXOS SOLVER follows.

```

NsProblemManager pm;

NsIntVar S(pm,1,9), E(pm,0,9), N(pm,0,9), D(pm,0,9),
          M(pm,1,9), O(pm,0,9), R(pm,0,9), Y(pm,0,9);

NsIntVar send = 1000*S + 100*E + 10*N + D;
NsIntVar more = 1000*M + 100*O + 10*R + E;
NsIntVar money = 10000*M + 1000*O + 100*N + 10*E + Y;

pm.add(send + more == money);

NsIntVarArray letters;
letters.push_back(S);
letters.push_back(E);
letters.push_back(N);
letters.push_back(D);
letters.push_back(M);
letters.push_back(O);
letters.push_back(R);
letters.push_back(Y);
pm.add(NsAllDiff(letters));

pm.addGoal(new NsgLabeling(letters));
if (pm.nextSolution() != false) {
    cout << "    " << send.value() << "\n"
         << " + " << more.value() << "\n"
         << " = " << money.value() << "\n";
}

```

If we execute the code, the result is

```

    9567
+  1085
= 10652

```

How do we state and solve a problem?

In the previous sections we stated some problems-examples. But what are the steps in order to state and solve another problem? Our code is summarized into the following triptych.

1. Constrained variables (`NsIntVar`) declaration, together with their domains

2. Constraints statement (`pm.add(·)`)
3. Goals declaration (`pm.addGoal(new NsgLabeling(·))`) and search for solutions (`pm.nextSolution()`)

The first thing to do is to create a problem manager (`pm`) to store the whole constraint network. The declaration is

```
NsProblemManager pm;
```

Next, we declare the constrained variables of the problem. Remember that while a simple variable (e.g. `int x`) stores only one value (e.g. `x = 5`), a *constrained* variable stores a *range* or, better, a domain. E.g., with the declaration `NsIntVar V(pm, 0, 5)`, the domain of `V` is the integer values range `[0..5]`.

When there are many constrained variables, then we use constrained variables arrays `NsIntVarArray`, as in the *N Queens* problem. E.g.

```
NsIntVarArray R;
```

The array `R` is initially empty. It is not possible to define a priori neither the array size, nor the included constrained variables domains. We can do this through an iteration

```
for (i = 0; i < N; ++i)
    R.push_back(NsIntVar(pm, min, max));
```

In place of `min` and `max` we put the minimum and maximum domain value, respectively. Next, we declare the existing constraints through `pm.add(·)` calls...

Before the end, if we solve an *optimization* problem, it remains to declare the parameter to optimize. When we find out this parameter-variable, we will pass it as an argument of `pm.minimize(·)`. This method is unnecessary when we seek for *any* solution of the problem.

We can now add a goal to be satisfied through the statement

```
pm.addGoal(new NsgLabeling(R));
```

This goal instructs the solver to assign values to the constrained variables of the array `R`. If we do not state this goal, the solver will not instantiate the variables `R[i]`, but it will only check the satisfaction of the constraints between *ranges*, and the variables will not become fixed.

Finally, we execute `pm.nextSolution()` to find a solution. This function is called inside a loop; every time it returns `true`, we have another unique problem solution.

If we have previously called `pm.minimize(·)`, the solver guarantees that each new solution will be better from the previous one. In case `pm.nextSolution()` returns `false`, then either the solution cost cannot be further improved, or there is not any other solution. Thus, we should have stored somewhere the last solution (and perhaps its cost too) in order to print it in the end, as in the following code for example

```

NsDeque<NsInt> bestR(N);

while (pm.nextSolution() != false) {
    // Record the current best solution
    for (i = 0; i < N; ++i)
        bestR[i] = R[i].value();
}

// Print the best solution...

```

2.4.2 Search methods as C++ classes

In order to facilitate or, better, to guide search, a *goals mechanism* has been implemented in the solver. The application developer that uses the solver can declare their own goals, or they can use the built-in ones. A goal often makes an assignment to a constrained variable, or it removes a value from the domain. If search reaches a dead-end, the solver automatically cancels the goals that guided to it, and the constraint network with its variables is restored back to the state before these goals were executed.

Generally speaking, a goal can assign or remove values to one or more variables, or it can be used to choose a variable in order to be successively assigned a value. In this way it defines the search method. While a goal terminates, it can optionally generate another goal; this possibility provides recursion characteristics to the goals mechanism. Last but not least, there are also the AND and OR *meta-goals*. They are called “meta-goals” because each of them is used to manipulate two *other* goals, namely *subgoals*. An AND-goal succeeds if its two subgoals succeed both, while an OR-goal succeeds if one or more of its subgoals succeed.

It is worth to mention that the OR-goals are also known as *choice points*. Indeed, they are points where we have two alternatives, that is points where the search tree branches off. During the execution of an OR-goal, its first subgoal is chosen, and if it finally fails, the solver cancels all the chain modifications that were made on the domains of the variables (after the first subgoal execution); the second subgoal is then tried. If the second subgoal also fails, then the whole OR-goal fails.

Object-oriented modelling

The declaration for the basic goal class in NAXOS SOLVER follows.

```

class NsGoal {
public:
    virtual bool isGoalAND(void) const;
    virtual bool isGoalOR(void) const;
    virtual NsGoal* getFirstSubGoal(void) const;
    virtual NsGoal* getSecondSubGoal(void) const;
    virtual NsGoal* GOAL(void) = 0;
};

```

The `NsgAND` and `NsgOR` meta-goal classes derive from the above `NsGoal` class. `NsgAND` and `NsgOR` constructor functions take two arguments (of `NsGoal*` type) that represent their two subgoals. Every `NsGoal` member-function—apart from `GOAL()`—has to do with meta-goals. The application developer that wants to define their own goals, has to take only care of `GOAL()`.

Every custom goal defined by the application developer should be a class that (directly or indirectly) extends `NsGoal`. Subsequently, function `GOAL()` should be defined in every goal class.

`GOAL()` is a critical method, as the solver executes it every time it tries to satisfy a goal. This method returns a pointer to another `NsGoal` instance, i.e. it returns the next goal to be satisfied. If the pointer equals to 0, this means that the current goal succeeded (was satisfied) and thus no other goal has to be created.

Therefore, an example follows, illustrating goals already built in the solver, as they are widely used. These goals describe the search method *depth-first-search* (DFS).

```
class NsgInDomain : public NsGoal {
private:
    NsIntVar& Var;

public:
    NsgInDomain(NsIntVar& Var_init)
        : Var(Var_init) { }

    NsGoal* GOAL(void)
    {
        if (Var.isBound())
            return 0;
        NsInt value = Var.min();
        return (new NsgOR(new NsgSetValue(Var,value),
                        new NsgAND(new NsgRemoveValue(Var,value),
                                   new NsgInDomain(*this))));
    }
};

class NsgLabeling : public NsGoal {
private:
    NsIntVarArray& VarArr;

public:
    NsgLabeling (NsIntVarArray& VarArr_init)
        : VarArr(VarArr_init) { }

    NsGoal* GOAL(void)
    {
        int index = -1;
        NsUInt minDom = NsUPLUS_INF;
        for (NsIndex i = 0; i < VarArr.size(); ++i) {
            if (!VarArr[i].isBound() && VarArr[i].size() < minDom) {
                minDom = VarArr[i].size();
            }
        }
    }
};
```

```

        index = i;
    }
}
if (index == -1)
    return 0;
return (new NsgAND(new NsgInDomain(VarArr[index]),
                  new NsgLabeling(*this)));
}
};

```

Regarding the practical meaning of the example, when we ask the solver to satisfy the goal `NsgLabeling(VarArr)`, we expect that all the variables of `VarArr` will be assigned values. Thus, the function `GOAL()` of `NsgLabeling` chooses a variable (specifically, the one with the smallest domain size according to the fail-first heuristic). Then it asks (via the goal `NsgInDomain` that assigns to a variable its domain minimum value) to instantiate the variable *and* to satisfy the goal `this`. This goal—that refers to a kind of “recursion”—constructs another `NsgLabeling` instance, that is identical to the current one. In fact, `this` tells the solver to assign values to the rest of `VarArr` variables. When `GOAL()` returns 0, we have finished.

`NsgLabeling` chooses the next variable to be instantiated, and `NsgInDomain` chooses the value to assign to this variable. More specifically, it always chooses the minimum value of the domain of the variable. Then it calls the built-in goal `NsgSetValue` that simply assigns the value to the variable. If it is proved afterwards that this value does not guide to a solution, it is removed from the domain by the goal `NsgRemoveValue`, and another value will be assigned by `NsgInDomain(*this)`.

Usually, when we face difficult and big problems, we should define our own goals, like `NsgLabeling` and `NsgInDomain`. The aim is to make search more efficient by using heuristic functions to take better decisions and choices tailored to specific difficult CSPs.

Variable and value ordering heuristics

The decisions about which variable to instantiate next and which value to assign to it are called *heuristics*, and they are crucial to the efficiency of a search method.

In the above implementation of the DFS goals, we chose *fail-first* as the *variable ordering heuristic*, which means that we prefer to instantiate first the variable having the minimum domain size. Another variable ordering heuristic known as *degree* is to choose the variable with the maximum edges (connections to other variables via constraints) in the constraint network.

Having chosen a variable, the next decision to make involves the *value ordering heuristic*. In the above DFS goals, we always select the minimum value out of the domain of a variable. This is also called a *lexicographical* ordering.

Such kind of ordering is fast but dummy. Alternatively, one could choose the value that is consistent to the maximum number of values belonging to the other variables.

3. RELATED WORK

In my own field, it once was possible for a grad student to learn just about everything there was to know about Computer Science. But those days disappeared about 30 years ago.

Donald Knuth, Things a computer scientist rarely talks about, 2001

Having introduced Constraint Programming and CSPs, we are now going to step into techniques found in the bibliography that are more closely related to our contributions that will follow this chapter (namely contributions in Heuristics and Search in Chapter 4, Distribution in Chapter 5, Propagation in Chapter 6).

3.1 Heuristics exploitation in related work

In the road to find a solution to a CSP, Constraint Programming solvers usually interchange search methods and constraint propagation.

- Search methods define the strategy of assigning values to the variables.
- Each time an assignment is made, constraint propagation assures that the other variables and their domains support the assignment.

Heuristics come into play when a search method has to decide (i) which *variable* is going to be instantiated next and (ii) which *value* out of its domain to assign to the variable. In the next two sections we describe important variable and value ordering heuristics.

3.1.1 Variable ordering heuristics

The most known variable ordering heuristic is the *fail-first* one, also known as *minimum remaining values* (MRV) or *most constrained variable* heuristic. It suggests that the variable having the minimum size of domain should be instantiated first. Nevertheless, more sophisticated approaches have been developed so far [45].

Impact-based search

The idea behind the *Impact-Based Search* (IBS) heuristic is to instantiate first the variable that will guide to the removals of as many values as possible out of

the domains of the other variables due to constraint propagation. More formally, this heuristic tries to minimize the Cartesian product of the domains $D_1 \times D_2 \times \dots \times D_n$ when propagation occurs due to an assignment.

A straightforward implementation of this is to consecutively assign each value $v \in D_x$ to the variable X and record the impact of each assignment after constraint propagation takes place. This should be done for every variable, and an aggregate score for each variable will be created. The variable with the highest score would be chosen for instantiation.

Propagating constraints without actually having decided to make an assignment is rather costly. What IBS proposes is to record the *history* of the impact of every previous assignment and expect that they will have the same impact in the future. Conclusively, IBS does not propagate constraints by itself, but it uses the statistics of the past constraint propagations.

Activity-based search

The variable ordering heuristic of *Activity-Based Search* (ABS) exploits even more statistics gathered during constraint propagation. For each search tree node, when a variable is affected by constraint propagation, its relevant score is increased. Else, if it remains unaffected, its score is decreased by a given factor. The variables with the highest scores are favored and will be the next to be instantiated.

The dom/wdeg heuristic

While the heuristics of IBS and ABS keep metrics for each value and each variable respectively, the *dom/wdeg* heuristic keeps statistics for the constraints. Each constraint c has a weight $w(c)$ which is equal to 1 plus the number of times that the constraint c could not be satisfied.

Furthermore, a weighted degree $wdeg(x)$ metric is defined for each constrained variable x as the sum of all $w(c)$ where c is a constraint that involves the variable x plus at least one more unassigned constrained variable.

Having defined the above, the *dom/wdeg* variable ordering heuristic favors the constrained variable x with the smallest $|D_x|/wdeg(x)$ ratio to be instantiated first.

A recent improvement of this heuristic suggests that each time we cannot satisfy a constraint c , we do not just increase the $w(c)$ weight by 1 but by a factor that depends on the number of the unassigned constrained variables that c involves and their current domain sizes [95].

Conflict history search

All the above heuristics did not consider involving *time* in their scores. Habet and Terrioux recently had the idea not only to record the failures of the constraints, but also to compute an *exponential recency weighted average* (ERWA) of these failures [45]. This means that during their *Conflict History Search* (CHS), we do not just accumulate failures of constraints, but we stress the importance of the constraints that tend to fail more recently.

How much recent constraint failures are favored? This depends on the factors of the ERWA formula. Depending on these factors of this moving average, we can favor more or less the recency of the failures. The ideal ERWA factors depend on the CSP parameters and cannot be known before solving it.

Here comes the *multi-armed bandit* (MAB) framework that allows us to use many different ERWA factors on the fly and exploit the more efficient ones [20].

Combinations using the multi-armed bandit framework

Imagine that you are in a casino and you have to choose to play among four slot machines. In fact, you cannot know a priori which one will give you back the most money.

This is the situation regarding the aforementioned heuristics; no one is a clear winner. In such cases, the *multi-armed bandit* (MAB) framework can be employed. It is a strategy to experiment with the arms of many “slot machines” and maximize your reward.

Xia and Yap recently used MAB to choose between IBS, ABS, and *dom/wdeg* heuristics at each search tree node [99]. Furthermore, Watez et al. also included CHS as one more arm to choose from. In their work, each heuristic-arm is chosen when search restarts and not on a search tree node level [94].

3.1.2 Value ordering heuristics

Least-constraining value (LCV) is a well-known value ordering heuristic. It suggests choosing the value (to assign to a constrained variable) that will trigger the removal of as few as possible values from the domains of the other variables during constraint propagation.

Another value ordering heuristic specially designed for Constrained Optimization Problems (COPs) that has been recently proposed is to choose the value that will guide to the biggest improvement of the cost function [31].

3.1.3 Heuristics in deterministic search methods

In *constructive search*, one can build a solution either with a deterministic/systematic search method or by making one-by-one random assignments. Do these methods exploit heuristics and how?

To our knowledge, existing search methods such as *limited discrepancy search* (LDS) use heuristics only to *order* the possible assignments and do not exploit the *difference* of the one heuristic estimation to another, but only their *rank* [75]. For example, the *iterative broadening* method initially explores only a limited children’s number for each search tree node [42]. Of course, in its first iterations, it chooses to visit only the children with the highest ranks. *Credit search* [7] and *limited assignment number* (LAN) [8] are other deterministic methods that also take into account the rank of the heuristic estimations and not the heuristic values themselves.

Last but not least, there are also methods that make the assumption that *the heuristic function is more reliable as the search tree node depth increases*. E.g.,

depth-bounded discrepancy search (DDS) allows to override a heuristic estimation, only when we have not yet reached a specific search tree depth [91]. Finally, there are some methodologies that take into account two or more heuristic functions and *learn* as the search proceeds which heuristic is the best to use [100].

3.1.4 Heuristics in random search methods

On the other hand, stochastic search methods completely ignore heuristics, as they choose to make an assignment at random [48]. For example, *depth first search with restarts* traverses the search tree making random choices, and when a specific time limit is reached, it restarts from the beginning.

3.1.5 Local search methods

The aforementioned search methods belong to *constructive search*, as they build a solution from scratch, step by step, by assigning a value to a variable each time.

On the other hand, there are non-systematic indirect search methods, also known as *local search* methods, which assemble a *candidate* solution, and then try to fix it by repairing conflicting sets of variables and constraints. Local search iteratively tries to *repair* the candidate solution, in order to satisfy the constraints a posteriori [46]. This is especially useful when solving difficult CSPs, and we are therefore happy just to find a solution, without usually caring if all candidate solutions will be examined.

Stochastic local search makes a random repair action in each step. There are many other local search variants.

Hill climbing. A well-known variant is *hill climbing*, also known as *iterative improvement*. In each step, it changes only one variable assignment (1-exchange). Normally, we make the change which will reduce the violated constraints number as much as possible [25] and this is called the *min-conflicts* local search heuristic that has been successfully applied in scheduling and to a plenty of other problems [2].

Simulated annealing. The above practice is prone to be trapped into *local minima*. This means that we can end up in a candidate solution that cannot be improved by modifying only one assignment anymore. In this case, we have to escape the current local minimum by making a random step.

Simulated annealing methodology permits random steps to skip local minima while a parameter called *temperature* is high; as time passes by and temperature drops, the method becomes less tolerant in random steps, especially if their evaluation is poor [28, 49]. In this work we attempt to bring this (local search) approach in constructive search methods.

3.1.6 Heuristics and probabilities

When a search method has to choose which constrained variable to instantiate next or which value should be assigned to the variable, heuristics come in handy.

Heuristics are normally used to order the available choices: the choice with the highest rank is mostly favored. Nevertheless, there is related work where a choice with a lower rank can also be favored. This can be accomplished by transforming heuristic values into probabilities.

Heuristic-biased stochastic sampling

In 1996, Bresina transformed the heuristic ranks into *probabilities* via the so-called *heuristic-biased stochastic sampling* (HBSS) [17]. He provided a set of various decreasing functions $\text{bias}(r)$, e.g. $\frac{1}{r}$ or e^{-r} etc., that take a specific integer choice rank $r \in \{1, 2, \dots\}$ and return a number that corresponds to the probability of the choice to be selected.

Example 3. Suppose that we have to choose a value to assign to a constrained variable. For example, we may have to assign to the variable X a value out of its domain $D_X = \{v_a, v_b, v_c, v_d\}$. Which one is better?

A heuristic comes into play to evaluate the four choices. Let us say that the respective heuristic values are $h_a = 8$, $h_b = 9$, $h_c = 6$, and $h_d = 7$. Hence, the respective ranks of the choices are $r_a = 2$, $r_b = 1$, $r_c = 4$, and $r_d = 3$. According to the heuristic function the choice b prevails.

Normal search methods would choose always option b , i.e. to assign v_b to X . Nevertheless, as mentioned above the HBSS method would make this choice non-deterministic and assign *probabilities* to each choice.

In this example, if we define $\text{bias}(r) = \frac{1}{r}$, the respective probabilities for the choices a, b, c, d would be $\frac{1}{2}, \frac{1}{1}, \frac{1}{4}, \frac{1}{3}$, each one divided by $\sum \text{bias}(r)$. Thus, we have $P_a = 0.24$, $P_b = 0.48$, $P_c = 0.12$, $P_d = 0.16$. Again, it is more probable to make choice b , with a 48% probability. But the alternative choices have significant probabilities too.

Value-biased stochastic sampling

Cicirello and Smith improved HBSS by introducing the *value-biased stochastic sampling* (VBSS). The bias function now takes as argument the heuristic *value* itself [22].

Example 4. Let us recompute the probabilities for the four choices in Example 3, using the VBSS methodology this time. We just need to substitute the $\text{bias}(r_i)$ function in the above example with the h_i value itself.

Therefore, the corresponding probabilities would be computed as $h_i / \sum h_i$ and we will get the probabilities $P_a = 0.27$, $P_b = 0.30$, $P_c = 0.20$, $P_d = 0.23$.

Note that the probabilities to make one of the choices a and b are almost equal. This is due to the fact that h_a and h_b are almost equal too, and these values directly affect the respective probabilities.

Heuristic equivalence

On the other hand, Gomes et al. exploit the so-called *heuristic equivalence* to equate the choices with the highest heuristic values [43]. In this way, we can

exclude the choices with the lower heuristic values and select at random amongst the choices with the most prevailing values.

Example 5. Again, let us consider the heuristic values of Example 3 and recompute the probabilities for the respective choices using the heuristic equivalence this time. Also, let us suppose that we set a threshold value 7.5. Every heuristic value above the threshold will be considered as “high.”

Therefore, the heuristic values below 7.5 will be considered as low, and the corresponding choices c and d will be discarded, with zero probabilities.

The remaining choices a and b would be then selected with equal probabilities $P_a = P_b = 0.5$.

Skewed probability distributions

Random search methods use the uniform distribution to select between the candidate choices. Gracas et. al have recently employed the geometric and the triangular probability distributions instead, also known as skewed probability distributions or non-symmetric. They first sort the candidate choices according to a given criterion and then map each choice to an already given decreasing probability distribution. Therefore, the first candidate choice has the greatest probability to be selected and the last candidate solution has the lowest probability to be made [44].

3.2 Distributing Constraint Programming with MapReduce

In 2004, Jeff Dean and Sanjay Ghemawat published *MapReduce*, a framework initially designed for Google’s very large database [27]. This paradigm synchronizes a plethora of machines, so as to read the whole Internet archive and “mine” information as needed.

3.2.1 Mappers and reducers

The cooperation of so many machines-nodes is viable, as MapReduce adopts a specific data flow architecture. This implies that there are some restrictions, as broadcasting a message and sharing data between nodes are not permitted.

The available computers are divided into two groups: The *Mappers* process the input. Each Mapper is assigned a part of the input. Then, it emits tuples such as $\langle r_1, s_1 \rangle$, $\langle r_2, s_2 \rangle$, etc. Each r_i denotes the *key* field of the tuple, while s_i contains the rest of the tuple’s fields.

The second machines group is the *Reducers* which accept the tuples. Depending to r_i , each tuple $\langle r_i, s_i \rangle$ is directed toward a specific Reducer.

Example 6. A basic introductory MapReduce application is used for counting the occurrences of each word in a text file.

1. For example, let us say that a text file has the content “design is not just what it looks and it feels like; design is how it works” [90].

2. Let us say that a MapReduce system with two mappers and two reducers is responsible to count how many times each word appears in the text, e.g. design,2.
3. The text is split into parts, e.g. “design is not just what it looks and it feels like” and “design is how it works”.
4. Each part is assigned to a specific mapper. Let us say that the first mapper will process the first part and the second mapper will process the second part of the text.
5. The first mapper emits the tuples for the occurrences of the words of the first part, in the form $\langle word, n \rangle$, where n is the occurrences number of the *word*: $\langle design, 1 \rangle$, $\langle is, 1 \rangle$, $\langle not, 1 \rangle$, $\langle just, 1 \rangle$, $\langle what, 1 \rangle$, $\langle it, 2 \rangle$, $\langle looks, 1 \rangle$, $\langle and, 1 \rangle$, $\langle feels, 1 \rangle$, $\langle like, 1 \rangle$.
6. Concurrently, the second mapper emits the tuples that correspond to the second part: $\langle design, 1 \rangle$, $\langle is, 1 \rangle$, $\langle how, 1 \rangle$, $\langle it, 1 \rangle$, $\langle works, 1 \rangle$.
7. The first field of each tuple is its *key*. Two tuples with same keys are sent to the same reducer. Generally speaking, each reducer is responsible to process specific keys. In this example, the first reducer processes the keys having a first character in the range a–m and the second reducer processes the tuples with keys that start with n–z.
8. Each reducer outputs the aggregate occurrences for the words-keys that it is responsible to process. The first reducer outputs $\langle and, 1 \rangle$, $\langle design, 2 \rangle$, $\langle feels, 1 \rangle$, $\langle how, 1 \rangle$, $\langle is, 2 \rangle$, $\langle it, 3 \rangle$, $\langle just, 1 \rangle$, $\langle like, 1 \rangle$, $\langle looks, 1 \rangle$. The output of the second reducer is $\langle not, 1 \rangle$, $\langle what, 1 \rangle$, $\langle works, 1 \rangle$.

When someone administers a cluster of computers, it is obviously more easy and more secure to provide access to the machines via a MapReduce framework, rather than set up a network topology for the specific user’s needs and grant him/her with the necessary privileges to use it.

For fields like Constraint Programming, there may exist more efficient distributed architectures and environments, usually proprietary and unavailable to experiment with. Nevertheless, MapReduce prevails as a standard in parallel/distributed computation and this work is a step toward exploiting it and adapting to it in order to solve Constraint Satisfaction Problems (CSPs).

3.2.2 Applications

MapReduce is an established framework to efficiently manage thousands of processors to complete tasks in parallel, which would have finished in many days in a sequential environment. For example, it has been used to explore social network graphs [1, 40] and it has been effectively applied to mine medical information out of large search engines logs [41].

There are many benefits of employing MapReduce.

- A MapReduce system, like Hadoop for example, will allocate by itself the CPU cores and/or the machines in a cluster of computers. There is no worry to invoke the appropriate number of threads-workers or communicate with other PCs to send them work.
- From the above, it is obvious that MapReduce supports both parallel and distributed environments.
- It is more easy and secure for a cloud administrator to provide access to their machines through a MapReduce interface, instead of allowing the implementation of ad hoc communication strategies among the machines. In some cases, there is no alternative except from adopting MapReduce.
- The no-communication between mappers restriction makes MapReduce highly scalable and capable of utilizing huge data centers.

These are some of the reasons why we leverage on a plain MapReduce approach.

Constraint Programming has been employed to improve the coordination and job scheduling for the dozens of mappers and reducers spread across many processors and machines [63]. Nevertheless, our contribution focuses on how we can employ MapReduce to boost Constraint Programming and not vice versa.

3.2.3 In the battle against the pandemic

The scalability of MapReduce enables scientists and organizations to process terabytes of data coming from multiple sources in the pandemic era. The data may refer to

- the genome of the viruses and the proteins produced by them [26]
- infection and fatality measurements across the globe [64]
- human behaviors such as social distancing in public areas.

For example, there is a surveillance system designed to massively process live video streams from public areas and warn the citizens that do not keep the minimum distance needed between them [59]. Other researchers use Hadoop which is in turn based on MapReduce to process a large number of posts in social media and extract opinions and sentiments related to the COVID-19 virus [30].

3.2.4 OR-parallelism vs. CSP partitioning

Constraint Programming independent search phase can be parallelized in several fashions. In the so-called *OR-parallelism* the search space (tree) is partitioned, and each processor-worker is assigned the task of exploring a specific part. If the workers use different search methodologies, we have a *portfolio* OR-parallelism [37].

On the other hand, there is another kind of parallelism where each processor is responsible for a certain division of the CSP, i.e. a number of constraints or variables, and it may check the validity of its constraints division or enforce a level of consistency between its variables division. In this case, communication between processors is unavoidable in order to ensure the whole CSP consistency [80].

3.2.5 A MapReduce and CP combination and other related work

MapReduce is closer to OR-parallelism, as Mappers are not permitted to communicate with each other. Régin et al. introduced MapReduceCP, the first known MapReduce and Constraint Programming (CP) combination [78]. In rough lines, they break up the search space into many pieces; the many fragments allow the more balanced search space distribution over Mappers. The Reducers' role in MapReduceCP is trivial, as they simply “echo” the solutions from the Mappers.

Régin et al. showed that MapReduceCP is superior to *work-stealing* [21], a dynamic search space partitioning schema implemented in Gecode [36], in which each idle worker consecutively sends messages to other workers in order to acquire a part of their current job.

Work-stealing is a non-MapReduce methodology that has a significant communication overhead between its workers, and this is something *SelfSplit*, another parallel methodology, tries to mitigate [33]. SelfSplit deterministically labels the search tree nodes with different “colors” (tags) without necessarily visiting them. Afterwards, each worker is responsible to process only the nodes of a specific color. The lack of communication between the workers has its ups and downs, as if most of the nodes of the search tree are labeled “red” than “green,” then the work to be shared will be unbalanced.

Apart from the MapReduce perspective, CSPs have been also viewed as embarrassingly parallel problems [77, 79] by decomposing them into many smaller CSPs. This is achieved by partitioning the search space, i.e. by splitting the Cartesian product of the domains of the constrained variables.

3.2.6 Partitioning the search space

The normal MapReduce schema divides a large file into small pieces; each piece is read by a Mapper. In CP we have not files but a search space, i.e. the Cartesian product of the domains $D_1 \times D_2 \times \dots \times D_n$.

Generally speaking, if the size of each domain D_i is d , we can partition the search space into d^m different search spaces $\{v_1\} \times \{v_2\} \times \dots \times \{v_m\} \times D_{m+1} \times D_{m+2} \times \dots \times D_n$, where $v_i \in D_i$.

Example 7. If we have a CSP with the variables X_1, X_2, X_3 , and X_4 , with the corresponding domains $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3\}$, i.e. $d = 3$, we can partition the search space into e.g. $d^2 = 9$ divisions:

$\{1\} \times \{1\} \times D_3 \times D_4$	$\{2\} \times \{1\} \times D_3 \times D_4$	$\{3\} \times \{1\} \times D_3 \times D_4$
$\{1\} \times \{2\} \times D_3 \times D_4$	$\{2\} \times \{2\} \times D_3 \times D_4$	$\{3\} \times \{2\} \times D_3 \times D_4$
$\{1\} \times \{3\} \times D_3 \times D_4$	$\{2\} \times \{3\} \times D_3 \times D_4$	$\{3\} \times \{3\} \times D_3 \times D_4$

In MapReduceCP this is how the search space is distributed over the Mappers [78]. This way of splitting the search space/tree is a *top-down* approach. For example, in a complete binary search tree, if the tree is split in two parts, the top two subtrees will be chosen. This seems to be an ideally balanced choice, as a

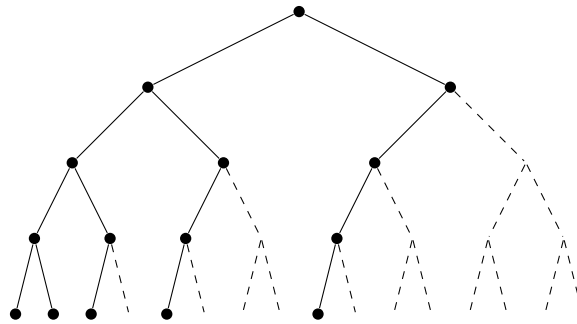


Figure 3.1: An asymmetric LDS binary search tree

complete tree is symmetric, and its two top left and top right subtrees are equally sized.

However, this is not always the case. Some search trees are unbalanced as the one in Fig. 3.1. There are at least two reasons for this asymmetry:

- Either some nodes are nogood, that is they do not lead to a solution because they violate a constraint, and there is no need to explore them further,
- or the search method is not complete and decides on purpose *not* to visit every search tree node.

Limited Discrepancy Search (LDS) is an example of a constructive search method that initially traverses a search tree like the one in the above figure [75].

In Chapter 5, our contribution will be to split any search tree (traversed by ad hoc search methods) by predicting/simulating the search tree topology, without actually traversing it. The generated partitions are then driven into a MapReduce system, which is today widespread in modern cloud infrastructures.

3.3 Constraint propagation related work

At the beginning, given a specific CSP, one would normally like to make sure if it has any solution or not. This information will be available only by using a standard *constructive* search method. As, in the worst case, these methods may exhaust all the candidate solutions of a CSP, it is important to make them more intelligent and prune the search space.

Starting from the 70's, the research on constraint propagation goes hand in hand with Constraint Programming research [56, 92]. Throughout all these years, there is a trend to invent stronger and stronger constraint propagation methodologies.

There are propagation methodologies tailor-made for local search [68]. Nevertheless, the focus of constraint propagation research is on constructive search methods.

3.3.1 Learning from mistakes or preventing them?

Look back techniques in backtracking search methods aim to avoid repeating the invalid assignments of the past. *Backjumping* is a well-known look back

technique, but it is not used in solvers, as other techniques clearly outperform it even in simple CSPs [3]. *Nogood learning* is a more promising look back technique that, based on the invalid assignments of the past, adds new constraints to avoid the invalid combination of assignments in the future [89]. However, these new constraints have the drawback of making the constraint network increasingly complex.

On the other hand, *look ahead* techniques are more proactive in the sense that they remove values out of the domains of the constrained variables *before* reaching an inconsistent assignment. *Maintaining arc consistency* (MAC) during search is the queen of all look ahead techniques [11]. According to MAC, each assignment to a domain of a variable is followed by an arc consistency enforcement method, such as the known optimal AC-2001 algorithm [12]. The optimality of AC-2001 was proven for enforcing arc consistency after a single assignment. But when we call repeatedly AC-2001 during search, after each single assignment, in order to *maintain* arc consistency, there is still room for improvements [54].

3.3.2 The importance of arc consistency

Arc consistency also plays a key role in splitting the CSPs into two large categories [11].

1. The *tractable* ones that can be solved in polynomial time, simply by maintaining arc consistency.
2. The *intractable* ones that are NP-complete problems and require an exponential backtracking algorithm to prove whether they have a solution or not.

Related work has defined the properties of the constraint network that suffice to categorize a CSP as tractable or intractable [24]. Furthermore, it has been recently proven that a CSP is tractable only if it contains specific types of constraints [18, 102].

In our work, for the sake of simplicity, we consider arc consistency only for binary constraints. The extension of arc consistency for constraints involving more than two variables is called *generalized arc consistency* (GAC). Contrary to conventional wisdom, there are studies that we can transform non-binary constraints into binary ones and enforce plain AC to them without losing the efficiency of GAC [93].

3.3.3 Higher-level consistencies (HLCs)

Arc consistency filters many futile values out of the domains of the constrained variables. But there are even stronger consistency levels than arc consistency.

These are the so-called *higher-level consistencies* (HLCs) and, while AC examines one constraint at a time, HLCs consider two or more constraints simultaneously. This makes them too expensive to be used in practice [5]. To mitigate the HLC overhead, there are hybrid strategies that go back and forth from HLC to AC [98]. Even machine learning has been employed to dynamically choose which consistency level is more efficient [4].

Table constraints: A prominent field for HLCs

Binary CSPs are very useful for theoretical analysis, as they are defined simply and every non-binary CSP can be converted into a binary one [81]. The analogue for the constraints is the so-called *table constraints*, in the sense that they are also simply defined, and every non-table constraint can be converted into a table constraint.

A table constraint is nothing more than the literal statement of a constraint as in Definition 1. For example, let the variables X and Y have the domains $D_X = D_Y = \{1, 2, 3, 4\}$. The constraint $X \neq Y$ is an “implicit” constraint that can be transformed into an “explicit” table constraint of the form $(X, Y) \in \{(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (3, 4), (4, 1), (4, 2), (4, 3)\}$. Admittedly, the latter is a clumsy way to state a constraint, not to mention the complexity overhead and counterintuitiveness while implementing it as a computer software.

On the other hand, from a theoretical and mathematical point of view, table constraints are a useful tool to group together every single kind of constraint. No need to apply a different validation or revision function for the constraints $X \neq Y$ and $Y = Z + 5$ for example.

There have been proposed tons of higher-level consistencies for table constraints in the bibliography [101]. Nevertheless, similarly to the binarization of CSPs, the “tabularization” of constraints is seldom applied in practice.

Back and forth to HLCs

Furthermore, the higher-level consistencies (HLCs) themselves are rarely used in practice for two main reasons.

- HLC methodologies require extra implementation effort in order to be used in Constraint Programming solvers, and they are usually complicated.
- HLC is not only complex in terms of implementation, but it also costs in terms of time. It is not a secret anymore that if we compare AC vs. HLCs for a wide range of problems, AC is usually faster [86].

According to recent related works, the remedy to mitigate this unexpected behavior is to employ an HLC instead of AC on the fly, only under specific conditions. In this context, Kostas Stergiou has classified all the so-called *adaptive constraint propagation* methodologies into three categories: *node*, *variable*, and *value-oriented* adaptive propagation [86].

Node-oriented adaptive propagation. While we traverse the search tree, node-oriented adaptive propagation methods decide separately for each search tree node if they are going to enforce arc consistency or another consistency level.

Variable-oriented adaptive propagation. While the AC vs. HLC decision in the node-oriented adaptive propagation algorithms affects all the constrained variables each time, the variable-oriented adaptive methodologies decide separately for each variable the consistency level that this variable will have towards the other variables.

Value-oriented adaptive propagation. Finally, if we take a separate decision for each value of each variable regarding which propagation algorithm we will use in order to seek a support for it, then our adaptive methodology is classified as value-oriented.

3.3.4 Toward more relaxed consistencies

In this work, we are moving toward the opposite direction, and instead of inventing an HLC and then trying to enforce it in practice only under certain conditions, we invest on bounds consistency (BC), a looser consistency level than arc consistency (AC).

We do not change consistency levels on the fly. We stick to one consistency level at a time (AC or BC) in order to keep the overall search algorithm that maintains consistency as simple as possible. This enables us to shed a more theoretical light to the integration of consistency into search and study the overall consistency complexities, not isolated but always *in the context of search methods* that maintain them. Our computations are backed by wide experimental data.

Instead of swapping HLCs and AC, we choose AC and BC, as bounds consistency is naturally used to describe constraints in Constraint Programming solvers [53].

3.3.5 Constraint propagation, validation, and explanation

It is almost never mentioned as it may seem trivial that constraint propagation also serves the role of constraint *validation*. For example, when a search method makes an assignment, constraint propagation attempts to prune as many nogood values as possible out of the domains of the constrained variables. If a domain is wiped out, then the assignment is considered invalid, and we have to proceed to a different assignment.

But, before moving forward to a new assignment, one may ask the solver to *explain* “why this assignment was proven invalid in the first place?” The answer to this question is useful both to a human and to a search algorithm. The user is informed which constraint fails, and if it fails frequently, the user may remove the constraint or make it less hard. On the other hand, a search method can record the explanation why a specific combination of assignments (e.g. $X \leftarrow a, Y \leftarrow b, Z \leftarrow c$) causes a domain wipeout (due to specific constraints that connect them). This can be a lesson learned for the search method, so as to avoid the same assignment in the future [29].

Besides, the trend now in Artificial Intelligence is to make it *explainable*: the so-called “XAI.” These days, intelligent algorithms play a key role into our lives. They should not be black boxes and we need justifications behind their decisions. In this context, the *Inverse Constraint Programming* has been recently introduced [50]. This framework does not just inform the user why a CSP cannot be solved, but it also suggests the needed changes in the user’s requirements in order to get a solution.

Furthermore, significant effort has been recently put into making the explanations as much user-friendly as possible. As constraint propagation is frequently a

big chain of steps and domain modifications, Bogaerts et al. defined the problem of finding the minimal sequence of steps that led to a failure [13]. This is actually an optimization problem by itself, to present to the user the smallest needed information about the decisions taken.

4. BRIDGING THE GAP: FROM RANDOM TO DETERMINISTIC HEURISTICS

This is the essence of intuitive heuristics: when faced with a difficult question, we often answer an easier one instead, usually without noticing the substitution.

Daniel Kahneman, Thinking, Fast and Slow

4.1 New probabilistic heuristics

Our contribution lies in the mathematical foundation of a framework that covers both deterministic and random heuristics in constructive search. In contrast to existing methodologies, we leverage on the *smooth* transition from the total randomness to determinism [71, 72].

4.1.1 Heuristic estimation as a real number

A heuristic function maps every possible choice in the search tree to a number that corresponds to the estimation that it will eventually guide us toward a solution.

Definition 5. For a specific search tree node, let Choices be the set with the alternative assignments that one may follow. The *heuristic function* h_i maps each alternative assignment $i \in \text{Choices}$ to a positive number, i.e. $h : \text{Choices} \rightarrow \mathbb{R}^+$.

Example 8. In Fig. 2.7 uppermost right node, there are two alternative assignments in $\text{Choices} = \{X_2 \leftarrow 1, X_2 \leftarrow 3\}$. One heuristic function may provide the estimations, e.g. $h_{X_2 \leftarrow 1} = 0.7$ and $h_{X_2 \leftarrow 3} = 2.8$; that is, the assignment $X_2 \leftarrow 3$ is more promising.

The above example is almost ideal, as the heuristic function h favors the assignment $X_2 \leftarrow 3$ over $X_2 \leftarrow 1$. Besides, the latter leads to a dead end, as its two descendants are struck-out in Fig. 2.7, because they violate the constraints.

Unfortunately, this is not always the case, i.e. the heuristic value for an assignment that leads to a dead end (say $X_2 \leftarrow 1$ in Fig. 2.7) may be overestimated or, even worse, may be greater than the heuristic estimation for an assignment that really leads to a solution (e.g. $X_2 \leftarrow 3$).

A heuristic value h_i is actually a *prediction* whether a specific assignment will ultimately guide us to a solution or not. Being a prediction, it implies an inherent *reliability/confidence* level.

In the above definition, we excluded negative values as the heuristic function's output. A negative heuristic evaluation could probably mean "do not make this

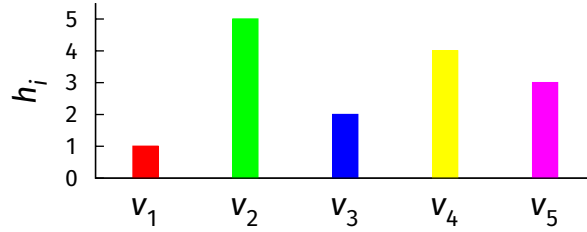


Figure 4.1: Heuristic estimations h_i for each value v_i

choice at all.” But heuristics are normally used to favor one choice over another and not to prune a choice. In any case, if we had a function h with $\min h < 0$, we could transform it into $h' = h + |\min h|$ to make it comply with the above definition.

4.1.2 Heuristics probabilistic foundations

Probabilities are a more precise way to depict heuristics than orderings, because heuristics are actually *estimations* whether a choice will guide us to a solution; they are not a strict quality rank.

Definition 6. A function $P : \text{Choices} \rightarrow [0, 1]$, namely a *heuristic distribution function*, maps each available choice to a corresponding probability, i.e. $P(i)$.

As in Definition 5 and the Example 8 that follows it, Choices may include all the possible/candidate assignments to a constrained variable.

Property 1. It should hold that $\sum_i P(i) = 1$, as P denotes a probability for each $i \in \text{Choices}$.

Regarding random search methods (Section 3.1.4), the probability is distributed uniformly along the Choices. Conclusively,

Property 2. The heuristic distribution for a random method is always $P(i) = \frac{1}{|\text{Choices}|}$, $\forall i$.

Example 9. Say that $\text{Choices} = \{v_1, v_2, \dots, v_5\}$. Every v_i denotes a possible assignment. Furthermore, in a specific search tree node we can make five different assignments, and their corresponding heuristic estimations h_i are 1, 5, 2, 4, 3 respectively, as in Fig. 4.1.

Figure 4.2 depicts the corresponding heuristic distribution function for a random method, that is $P(i) = \frac{1}{5}$, $\forall i$.

On the other extreme, deterministic search methods (Section 3.1.3) always select the choice v_i that corresponds to the h_i with the highest rank.

Property 3. Formally, in deterministic search methods, if $i = \arg \max_j h_j$, then $P(i) = 1$, otherwise $P(i) = 0$.

Example 10. The greatest heuristic value in Example 9 is $h_2 = 5$. Hence, a deterministic search method would select v_2 with a certain probability $P(2) = 1$. Consequently, the rest of the probabilities are zero, as in Fig. 4.3.

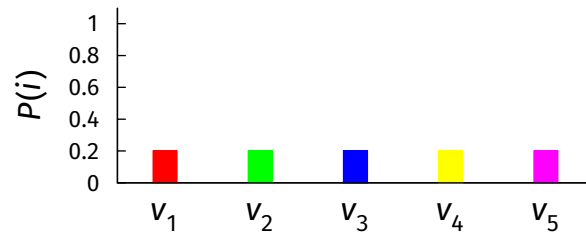


Figure 4.2: The probability is spread uniformly

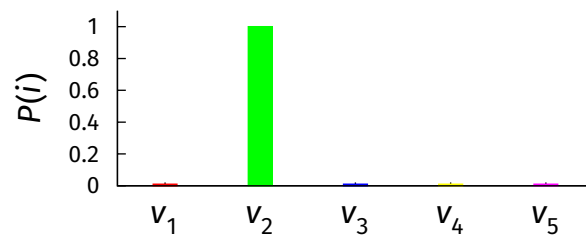


Figure 4.3: Systematic search favors the highest h_i

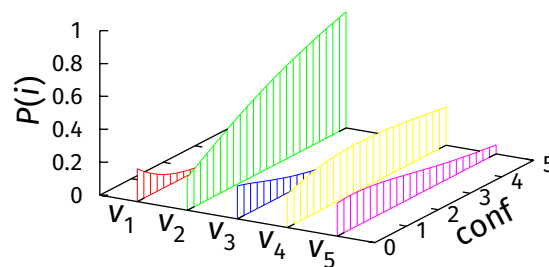


Figure 4.4: As conf rises, the effect to $P(i)$ is greater

Without loss of generality, we assume that heuristic values are non-zero. Furthermore, if there are more than one heuristic value equal to the maximum heuristic value, deterministic methods arbitrarily consider only one of them as maximum. To simplify the following equations, we will assume that there is only one maximum. For formulas and proofs that are straightforwardly compatible with two or more equal maximum heuristic values, see Section 4.1.4.

4.1.3 Bridging the two opposites

We extend our previous formulation of the heuristic distribution function (Definition 6) in order to compromise random and deterministic methods. We introduce a parameter $\text{conf} \in \mathbb{R}^+$, that signifies how much the heuristic estimations will be taken into account; it is the heuristic's *confidence*. This confidence parameter is the basis to define the condition when a heuristic distribution function is “balanced.”

Definition 7. A parameterized heuristic distribution function $P_{\text{conf}}(i)$ is *balanced* if and only if:

1. $\forall i, \lim_{\text{conf} \rightarrow 0} P_{\text{conf}}(i) = \frac{1}{|\text{Choices}|}$, and
- 2a. if $i = \arg \max_j h_j$, $\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = 1$,
- 2b. otherwise, $\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = 0$.

Moreover, the function $P_{\text{conf}}(i)$ must be monotonic and continuous with respect to conf and for fixed i .

Intuitively, conf is the link between random and deterministic search methods, as the above definition covers both Property 2 when $\text{conf} \rightarrow 0$ and Property 3 when $\text{conf} \rightarrow \infty$. In other words, conf is the position along the random-deterministic axis.

What happens for intermediate conf values? This depends on the precise parameterized heuristic distribution function instance. We define the following function that gradually scales randomness.

Lemma 1. The function $P_{\text{conf}}(i) = \frac{h_i^{\text{conf}}}{\sum_j h_j^{\text{conf}}}$ is *balanced*.¹

Proof. We prove Definition 7 three requirements.

$$1. \lim_{\text{conf} \rightarrow 0} P_{\text{conf}}(i) = \frac{h_i^0}{\sum_{j \in \text{Choices}} h_j^0} = \frac{1}{\sum_{j \in \text{Choices}} 1} = \frac{1}{|\text{Choices}|}.$$

- 2a. Let $n = |\text{Choices}|$. This number is bounded as the possible assignments in a CSP are a finite set. Thus, the distribution function can be analyzed as

$$P_{\text{conf}}(i) = \frac{h_i^{\text{conf}}}{\sum_j h_j^{\text{conf}}} = \frac{h_i^{\text{conf}}}{h_1^{\text{conf}} + h_2^{\text{conf}} + \dots + h_{\text{max}}^{\text{conf}} + \dots + h_n^{\text{conf}}}.$$

¹For $\text{conf} = 1$, the function $P_1(i) = \frac{h_i}{\sum_j h_j}$ is equivalent to the *fitness proportionate selection* function—resembling a *roulette wheel*—that is used in Genetic Algorithms [84].

Let h_{\max} be the maximum h_j . If we divide by h_{\max}^{conf} both the nominator and denominator, we have

$$\begin{aligned} P_{\text{conf}}(i) &= \frac{\left(\frac{h_i}{h_{\max}}\right)^{\text{conf}}}{\left(\frac{h_1}{h_{\max}}\right)^{\text{conf}} + \dots + 1 + \dots + \left(\frac{h_n}{h_{\max}}\right)^{\text{conf}}} \\ &= \frac{\left(\frac{h_i}{h_{\max}}\right)^{\text{conf}}}{1 + \sum_{j \neq \max} \left(\frac{h_j}{h_{\max}}\right)^{\text{conf}}}. \end{aligned} \quad (4.1)$$

Here, \max is an abbreviation for $\arg \max_j h_j$. Therefore, $\forall j \neq \max$,

$$\begin{aligned} h_j < h_{\max} &\Rightarrow \frac{h_j}{h_{\max}} < 1 \Rightarrow \\ &\lim_{\text{conf} \rightarrow \infty} \left(\frac{h_j}{h_{\max}}\right)^{\text{conf}} = 0. \end{aligned} \quad (4.2)$$

As a result from (4.1) and (4.2),

$$\begin{aligned} \lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) &= \frac{\lim_{\text{conf} \rightarrow \infty} \left(\frac{h_i}{h_{\max}}\right)^{\text{conf}}}{1 + \sum_{j \neq \max} \lim_{\text{conf} \rightarrow \infty} \left(\frac{h_j}{h_{\max}}\right)^{\text{conf}}} \\ &= \lim_{\text{conf} \rightarrow \infty} \left(\frac{h_i}{h_{\max}}\right)^{\text{conf}}. \end{aligned} \quad (4.3)$$

A direct derivation of the above is that for $i = \max \equiv \arg \max_j h_j$, we have $\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(\max) = 1$, which is the second prerequisite for a balanced function.

- 2b. Finally, the last prerequisite of Definition 7 involves $i \neq \max \Rightarrow h_i < h_{\max} \Rightarrow \frac{h_i}{h_{\max}} < 1$, which, combined with (4.3), gives $\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = 0$, which had to be demonstrated.

□

The above function in Lemma 1 is balanced, and it also moves smoothly from the random extreme to the deterministic one, because it is a *continuous* function, with regard to $\text{conf} \in \mathbb{R}^+$.

Hence, the overall function is a transition from the total randomness to the almost total determinism. This is illustrated in the three-dimensional Fig. 4.4, which for $\text{conf} = 0$, is equivalent to the two-dimensional Fig. 4.2, and when $\text{conf} \rightarrow \infty$, it is equivalent to Fig. 4.3.

4.1.4 Balanced heuristic distribution for two or more maximum heuristic values

This section is a restatement of the definition and proof of the previous section, for the case that there are m equal maximum heuristic values with $m > 1$.

Definition 8. We denote the set containing the indexes of the heuristic values with maximum values as $M = \{i \mid h_i = h_{\max}\}$, where h_{\max} is the maximum heuristic value. Also, by definition, $|M| = m$.

Definition 9. A parameterized heuristic distribution function $P_{\text{conf}}(i)$ is *balanced* if and only if:

1. $\forall i, \lim_{\text{conf} \rightarrow 0} P_{\text{conf}}(i) = \frac{1}{|\text{Choices}|}$, and
- 2a. if $i \in M, \lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = \frac{1}{m}$,
- 2b. otherwise, $\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = 0$.

Moreover, the function $P_{\text{conf}}(i)$ must be monotonic and continuous with respect to conf and for fixed i .

Lemma 2. The function $P_{\text{conf}}(i) = \frac{h_i^{\text{conf}}}{\sum_j h_j^{\text{conf}}}$ is balanced.

Proof. We prove the three requirements of Definition 9.

$$1. \lim_{\text{conf} \rightarrow 0} P_{\text{conf}}(i) = \frac{h_i^0}{\sum_{j \in \text{Choices}} h_j^0} = \frac{1}{\sum_{j \in \text{Choices}} 1} = \frac{1}{|\text{Choices}|}.$$

2a. Let $n = |\text{Choices}|$. This number is bounded as the possible assignments in a CSP are a finite set. Thus, the distribution function can be analyzed as

$$P_{\text{conf}}(i) = \frac{h_i^{\text{conf}}}{\sum_j h_j^{\text{conf}}} = \frac{h_i^{\text{conf}}}{\sum_{j \in M} h_j^{\text{conf}} + \sum_{j \notin M} h_j^{\text{conf}}}.$$

Let h_{\max} be the maximum h_i . If we divide by h_{\max}^{conf} both the nominator and denominator, we have

$$P_{\text{conf}}(i) = \frac{\left(\frac{h_i}{h_{\max}}\right)^{\text{conf}}}{\sum_{j \in M} \left(\frac{h_j}{h_{\max}}\right)^{\text{conf}} + \sum_{j \notin M} \left(\frac{h_j}{h_{\max}}\right)^{\text{conf}}}. \quad (4.4)$$

From the Definition 8 it holds that for each $j \in M, h_j = h_{\max}$. Therefore

$$\sum_{j \in M} \left(\frac{h_j}{h_{\max}}\right)^{\text{conf}} = \sum_{j \in M} \left(\frac{h_{\max}}{h_{\max}}\right)^{\text{conf}} = \sum_{j \in M} 1 = m. \quad (4.5)$$

Furthermore, for each $j \notin M$,

$$\begin{aligned}
 h_j < h_{\max} &\Rightarrow \frac{h_j}{h_{\max}} < 1 \Rightarrow \\
 \lim_{\text{conf} \rightarrow \infty} \left(\frac{h_j}{h_{\max}} \right)^{\text{conf}} &= 0.
 \end{aligned} \tag{4.6}$$

As a result from (4.4), (4.5), and (4.6),

$$\begin{aligned}
 \lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) &= \frac{\lim_{\text{conf} \rightarrow \infty} \left(\frac{h_i}{h_{\max}} \right)^{\text{conf}}}{m + 0} \\
 &= \frac{1}{m} \lim_{\text{conf} \rightarrow \infty} \left(\frac{h_i}{h_{\max}} \right)^{\text{conf}}.
 \end{aligned} \tag{4.7}$$

A direct derivation of this is that for $i \in M$, we have

$$\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = \frac{1}{m} \lim_{\text{conf} \rightarrow \infty} \left(\frac{h_{\max}}{h_{\max}} \right)^{\text{conf}} = \frac{1}{m}$$

which is the second prerequisite for a balanced function.

2b. Finally, the last prerequisite of Definition 9 involves $i \notin M \Rightarrow h_i < h_{\max} \Rightarrow \frac{h_i}{h_{\max}} < 1$, which, combined with (4.7), gives $\lim_{\text{conf} \rightarrow \infty} P_{\text{conf}}(i) = 0$, which had to be demonstrated. □

Furthermore, our initial goal was to propose flexible heuristics which perform better than purely deterministic or purely stochastic ones. To implement and measure the transition from randomness to determinism, we just introduced a *confidence* value. However, new questions now arise. Which conf value should be used? Which is the best way to bind the proposed hybrid heuristics to search processes?

4.2 Piece of Pie Search

Heuristics are not completely autonomous on themselves. Their usage is meaningful only in the context of search methods. Search methods consult/call heuristic functions and not the opposite. In order to fully exploit the introduced heuristics framework, we built the new constructive search method *Piece of Pie Search* (PoPS).

4.2.1 The algorithm's core

Figure 4.5 describes POPSAMPLE, which is the PoPS core. It is called inside PoPS in order to solve a CSP by providing a complete and valid Assignments

set, which is initially empty. For the sake of simplicity, we consider that the value $\text{conf} = 100$ represents infinity or, more formally, the maximum conf value. In fact, before “hard-coding” this value we should consider the heuristic function itself which, in turn, depends on the CSP that is used for and then probably change this “100” value.

In each `POPSAMPLE` call we get an unassigned variable returned by the function `VARIABLESORDERHEURISTIC(\mathcal{X})` where \mathcal{X} is the set of all the constrained variables. For example, if this function implements the fail-first heuristic, it will return the variable X having the minimum $|D_X|$ with $|D_X| > 1$.

The next step is to store its D_X domain in order to restore it in a future backtrack. All the above steps are common in constructive search methods.

The crucial and novel part of this function is inside the **while** iteration where we go through the different values in D_X . The `VALUESORDERHEURISTIC(D_X, conf)` call returns a value out of D_X using the heuristic function in Lemma 1.

Normal search methods, like Depth First Search (DFS), Limited Discrepancy Search (LDS), and other known deterministic methods explore in their steps a specific *number* of values in D_X or every value in it (cf. Section 3.1.3). In `POPSAMPLE`, we explore a specific *subset* D'_X of D_X , which corresponds to a proportion of the heuristics pie. The proportion is the argument `PieceToCover` $\in [0, 1]$. When `PieceToCover` is 1, `POPSAMPLE` becomes a complete search method as it explores all the D_X set values.

Example 11. Figure 4.6 demonstrates the heuristics-probabilities pie for the Example 9: Each $P(i)$ corresponds to a value v_i in D_X . In this case, a `POPSAMPLE(0.5, 1)` invocation would explore at least half the pie. E.g., the choices that correspond to the heuristics $P(1) + P(2) + P(3)$ or $P(2) + P(5)$ make half the pie and more.

A more detailed step by step explanation follows.

- We are inside the **while** loop of a `POPSAMPLE(0.5, 1)` call.
- `CoveredPiece` is initially 0; the loop stops when `CoveredPiece` exceeds 0.5.
- `VALUESORDERHEURISTIC($D_X, 1$)` is called.
- According to Example 9, the above function call will return a value out of $\{v_1, v_2, v_3, v_4, v_5\}$.
- Each value v_i has been evaluated with a heuristic value h_i .
- The h_i function may implement for example the so-called least constraining value (LCV) heuristic. Any heuristic function h_i can be used.
- Let us use the indicative values $h_1 = 1$, $h_2 = 5$, $h_3 = 2$, $h_4 = 4$, and $h_5 = 3$.
- The probability that v_i is selected by `VALUESORDERHEURISTIC` is $P(i)$, which is calculated using the above evaluations together with Lemma 1.
- Thus, the respective probabilities are $P(1) = 0.07$, $P(2) = 0.33$, $P(3) = 0.13$, $P(4) = 0.27$, and $P(5) = 0.20$.
- Again, all the above are *probabilities* ($P(i)$) of the event that a specific value (v_i) will be selected. Therefore, every value can be selected in each iteration.

```

function POPSAMPLE(PieceToCover, conf)
  arguments:
    PieceToCover: The proportion of the heuristics' pie to be explored
    conf: A "confidence" value between 0 and 100
  local variables:
    Assignments: set with all the assignments made until this call
     $\mathcal{X}$ : set with all the constrained variables (bound and unbound)
    X: constrained variable that is going to be instantiated
    value: value that is going to be assigned
     $h_{X \leftarrow v}$ : heuristic value for the assignment  $X \leftarrow v$ 
     $D_{X_{init}}$ : initial domain of X, before any assignment was made
     $D_X$ : current domain of X
    CoveredPiece: current covered proportion of the pie

  if Assignments violate any constraint then
    return failure
  else if Assignments include every variable then
    Record Assignments as solution
    return success
  end if
   $X \leftarrow \text{VARIABLESORDERHEURISTIC}(\mathcal{X})$ 
   $D_{X_{init}} \leftarrow D_X$ 
  CoveredPiece  $\leftarrow$  0
  while CoveredPiece  $\leq$  PieceToCover do
    value  $\leftarrow \text{VALUESORDERHEURISTIC}(D_X, \text{conf})$ 
    CoveredPiece  $\leftarrow$  CoveredPiece +  $\frac{h_{X \leftarrow \text{value}}^{\text{conf}}}{\sum_{v \in D_{X_{init}}} h_{X \leftarrow v}^{\text{conf}}}$ 
    Assign value to X and add it to Assignments
    if POPSAMPLE(PieceToCover, conf +  $\frac{100 - \text{conf}}{|\mathcal{X}| - 1}$ ) then
      return success
    end if
    Undo the assignment
     $D_X \leftarrow D_X - \{\text{value}\}$ 
  end while
   $D_X \leftarrow D_{X_{init}}$   $\triangleright$  Restores initial domain
  return failure  $\triangleright$  All alternative values are exhausted
end function
    
```

Figure 4.5: The recursive POPSAMPLE called by PoPS

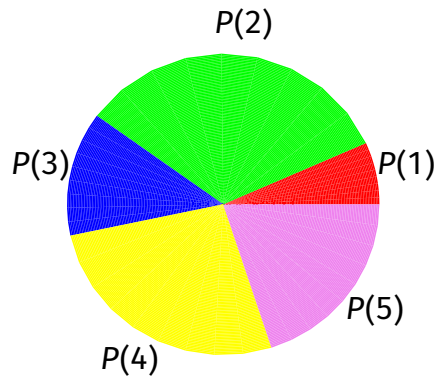


Figure 4.6: The heuristics-probabilities pie chart for Example 9

- Suppose that v_5 is selected at the first iteration with $P(5) = 0.20$.
- This probability is also used to increase the current *CoveredPiece*, which becomes 0.20 too.
- X is assigned v_5 .

After the assignment, $\text{POPSAMPLE}(0.5, 1 + \frac{99}{|\mathcal{X}|-1})$ is called. Please note the increase of the conf value. This recursive call will choose another variable out of \mathcal{X} and enter the **while** loop again. This loop will try to assign a value to the new variable from its domain. If all the attempts inside the iteration of the new recursive call fail, we continue back to the first **while** loop, which was described in the above bullets.

- The assignment of v_5 to X is undone, v_5 is removed from the domain, and another iteration begins.
- We proceed to the second iteration, as the *PieceToCover* (0.5) is still greater or equal than the *CoveredPiece* (0.2).
- Let us say that v_2 is then chosen by *VALUESORDERHEURISTIC* with a $P(2) = 0.33$ probability.
- *CoveredPiece* now equals $0.20 + 0.33 = 0.53$.

Then, $\text{POPSAMPLE}(0.5, 1 + \frac{99}{|\mathcal{X}|-1})$ is called. Again, if all the attempts in the iteration of the new call to instantiate the next variable fail, we step back to the first **while** loop:

- The assignment of v_2 to X is undone.
- We proceed to the third iteration.
- Nevertheless, *CoveredPiece* (0.53) is now greater than *PieceToCover* (0.5).
- More than half of the pie of the choices for X has been already explored; no other alternatives are examined.

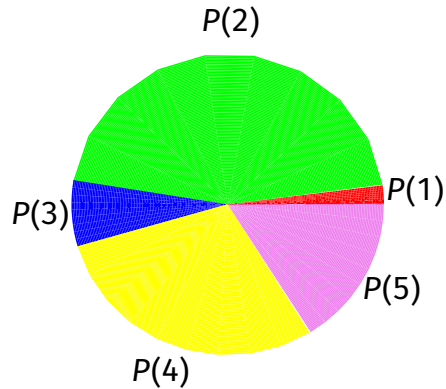


Figure 4.7: The previous heuristics-probabilities pie chart when conf = 2

- The rest of the values v_1, v_3, v_4 are left unused/unexplored. This makes POPSAMPLE an *incomplete* search method, as it may override a solution (which involves for example these values) for the sake of speed.

LDS is a search method which, in each search tree node, explores only a limited *number* of the available choices. The difference with our method is that we may explore a limited *proportion* of the heuristics pie of choices, which makes our method more “heuristics-aware.” This means that the number of the explored choices by our method in a specific node may vary, depending on how the heuristics pie is distributed to the choices. On the other hand, LDS explores a fixed number of choices, independently of the heuristics pie distribution.

It is worth noting that while more variables get instantiated, the conf value gradually increases. Besides, heuristic estimations tend to be more reliable when we have less unassigned variables.

Example 12. We will consider the above Example 11 for a POPSAMPLE(0.5, 2) call, i.e. for conf = 2.

According to Lemma 1, the probabilities for conf = 2 are computed as $P(i) = \frac{h_i^2}{\sum_j h_j^2}$. For example, $P(1) = \frac{h_1^2}{\sum_j h_j^2} = \frac{1^2}{1^2+5^2+2^2+4^2+3^2} = 0.02$. The other probabilities are $P(2) = 0.45$, $P(3) = 0.07$, $P(4) = 0.29$, and $P(5) = 0.16$. Thus, the pie is redistributed as in Figure 4.7.

While the conf value increases, the value v_2 which had initially the greatest heuristic evaluation h_2 is even more likely to be selected, as $P(2)$ increases too. In other words, we get closer to total determinism and closer to complete confidence in the highest heuristic evaluation: In total determinism (in systematic search) v_2 would have been always selected with a certain probability 1.

4.2.2 Heuristic confidence vs. node level

An important detail in POPSAMPLE appearing in Fig. 4.5, is the increase in conf as the current search tree node level deepens.

When we make the first recursive POPSAMPLE call (inside **while**), we have already made an assignment. Hence, the current tree level will be augmented by

1 and conf will be increased by $\frac{100-\text{conf}}{|\mathcal{X}|-1}$.

Each subsequent recursive call deepens search by one level, until the current depth reaches $|\mathcal{X}|$, which means that every variable in \mathcal{X} has been assigned a value. For a specific depth k the conf value is increased by $(k-1)\frac{100-\text{conf}}{|\mathcal{X}|-1}$. Finally, when $k = |\mathcal{X}|$, the conf argument of POPSSAMPLE will become equal to the value 100.

The following is not guaranteed, but in the deepest node levels, heuristics are *usually* more accurate, because more variables have been instantiated, and we have a clearer picture of the problem. In our framework, more accuracy means more confidence, that is why we increase conf as the search method proceeds with the assignments.

4.2.3 POPSSAMPLE average complexity

The POPSSAMPLE complexity depends on PieceToCover argument and the heuristic function distribution.

Lemma 3. *Let n be the constrained variables number and let d be the average domain size. Then, the average complexity of a POPSSAMPLE(PieceToCover, conf) call is $O(d^n \cdot \text{PieceToCover}^n)$.*

Proof. An initial POPSSAMPLE(PieceToCover, conf) call iterates through the values of, let us say, the first variable X_1 . If the heuristic function numbers for the values in D_{X_1} are uniformly distributed, the expected value for $h_{X_1 \leftarrow \text{value}}$ would be $\mu = \frac{\sum_{v \in D_{X_1}} h_{X \leftarrow v}}{|D_{X_1}|}$.

Thus, to reach the pie proportion $A = \text{PieceToCover} \cdot \sum_{v \in D_x} h_{X \leftarrow v}$, we need $A/\mu = \text{PieceToCover} \cdot |D_{X_1}|$ iterations, i.e. $O(\text{PieceToCover} \cdot d)$ loops.

The total time needed is $T_1 = O(\text{PieceToCover} \cdot d) \cdot T_2$, where T_2 is the time for the POPSSAMPLE call *inside* the loop. It also holds that $T_2 = O(\text{PieceToCover} \cdot d) \cdot T_3$, etc., and finally $T_n = O(\text{PieceToCover} \cdot d)$. In conclusion, the aggregate complexity is $O(\text{PieceToCover}^n \cdot d^n)$ for the initial call. \square

We can observe that POPSSAMPLE(1, conf) is equivalent to a complete search space exploration, which has an $O(d^n)$ time complexity.

4.2.4 The motivation behind PoPS

Finding the best conf is the motivation behind PoPS. Unfortunately, we do not know a priori which conf is the best parameter for POPSSAMPLE. However, we can find it by trial and error. In Fig. 4.8, the PoPS function invokes POPSSAMPLE for SamplesNum different conf_i values, including the values 0 and 100.

Each different conf_i is used in turn. Initially, the Cover _{i} parameter in the PoPS algorithm is zero for every conf_i . When a specific conf_i has been examined, the corresponding Cover _{i} is increased by $\frac{1}{d}$, where d is the average domain size. When the second iteration over a specific conf_i ends, the Cover _{i} is increased again by $\frac{1}{d}$ and so on.

```

function PoPS
  local variables:
    SamplesNum: how many different conf values are initially employed
    confi: array with all the initially employed conf values
    Samplei: a Boolean array; if its ith element is deactivated (false) the
               corresponding confi value is currently ignored
    Coveri: corresponding “piece to cover” argument for POPSSAMPLE call
    d: average domain size of the constrained variables

  for i from 1 to SamplesNum do
    Samplei is activated
    Coveri ← 0
    confi ← 100 ·  $\frac{i-1}{\text{SamplesNum}-1}$ 
  end for
  while the available time is not exhausted do
    for each active Samplei do
      if POPSSAMPLE(Coveri, confi) did not return a solution then
        Samplei is deactivated
      end if
      Coveri ← Coveri +  $\frac{1}{d}$ 
    end for
    if every Samplei is deactivated then
      Activate every Samplei    ▷ to keep searching.
    end if
  end while
end function

```

Figure 4.8: Piece of Pie Search (PoPS) Method

In this way, each conf_i is given the same opportunity (search space) to find a solution. If some conf_i does not produce a solution, it is deactivated. It is reactivated only if all other conf_i 's fail to produce a solution.

4.3 Empirical evaluations

The gradual switch from randomness to determinism can boost search in demanding CSPs, such as course scheduling and the radio frequency assignment problems. With the help of our free constraint programming C++ library NAXOS SOLVER [67], we solved official instances of these problems for different heuristic distribution configurations.

The source code for our evaluations is freely available, including the problem datasets.² The experiments were conducted on an HP computer with an Intel dual-core E6750 processor clocked at 2.66 GHz with 2 GB of memory and a Xubuntu Linux 12.04 operating system.

In the following three subsections (4.3.1, 4.3.2, 4.3.3) the experiments are repeated for different conf values, as we do not use PoPS. On the other hand, in the last subsection 4.3.4, PoPS automatically chooses by itself the employed conf values.

4.3.1 University course scheduling

Automated timetabling is nowadays a crucial application, as many educational institutions still use ad hoc manual processes to schedule their courses. The International Timetabling Competition (ITC) is an attempt to unify all these processes. We borrowed the fourteen instances of the contest track concerning *universities* [58].

In these problems, we have to assign valid teaching periods and rooms to the curriculum lectures. The objective is to distribute them evenly during the week but without having gaps between them, if scheduled on the same day; each gap increases the solution cost [74]. As *variable ordering heuristic*, we used minimum remaining values and degree for tie breaking, and we randomized it using the function in Lemma 1. Least constraining value was used as *value ordering heuristic*.

Due to the ITC specifications, we had 333 seconds in our machine to solve each instance and minimize the solution cost as much as we could. Figures 4.9 and 4.10 display the minimum solution costs found per instance for various conf values. We observe that as conf increases the costs tend to a specific number, whilst for small conf values we have fluctuations because search becomes more random.

It was expected that for high conf values the results would be more stable, as the search process approximates the default depth-first-search (DFS). For the marginal low values, e.g. $\text{conf} = 0$, search is completely stochastic and the results are worse on average, as we have higher solution costs. Nevertheless, the evaluations for intermediate conf values, e.g. $\text{conf} \approx 20$, are more promising.

²<http://di.uoa.gr/~pothitos/PoPS>

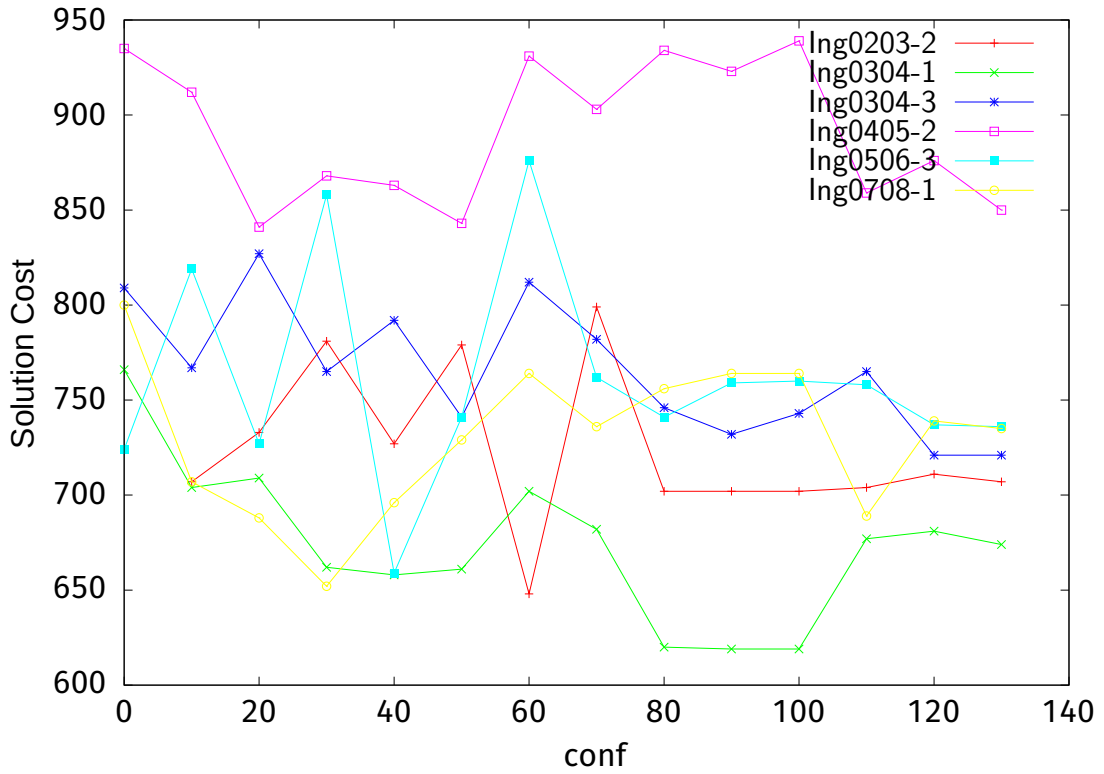


Figure 4.9: Timetabling solutions costs vs. conf

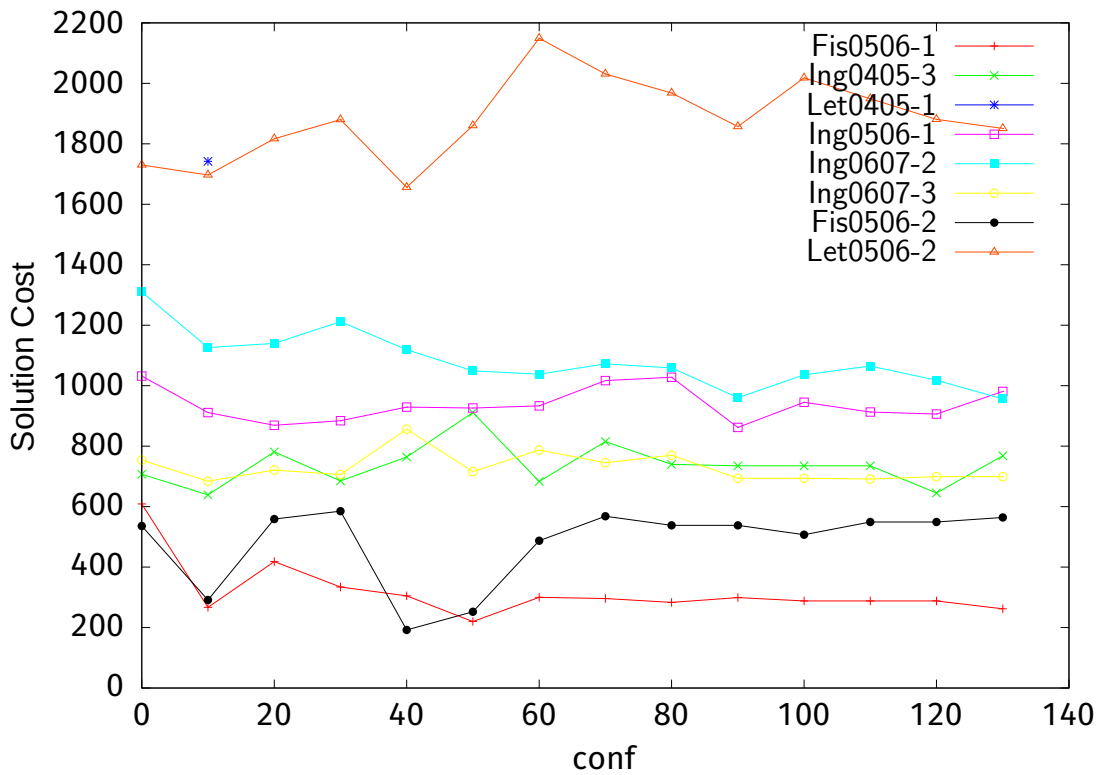


Figure 4.10: Solutions for the rest of the ITC instances

Remember that an intermediate *conf* value favors the selection which corresponds to the best heuristic evaluation, *but* it also gives room to other selections (the “outsiders”) as their probabilities are not zero.

The automatic selection of the best *conf* is an open question here; in Section 4.3.4, PoPS finds automatically the best *conf* values.

In practice, as shown in Fig. 4.9 and 4.10, a *conf* value around 100 actually represents infinity, because search tends to produce the same solutions for $\text{conf} \geq 100$.

It is worth to mention that in Fig. 4.10 the only solution found for the Let0405-1 instance, depicted with an asterisk $*$, was for an intermediate $\text{conf} = 10$.

4.3.2 Radio link frequency assignment

Another important real problem is the frequency assignment, in which we have to assign a frequency to each radio transmitter with the objective to minimize the interference. The interference is minimized by assigning different frequencies to every two transmitters that are close to each other.

The Centre Electronique de l’Armement (CELAR) provides a set of real datasets for this NP-hard problem [19]. We chose to solve the five so-called “MAX” problem instances, namely SCEN06–SCEN10, in which, generally speaking, we try to maximize the number of the satisfied soft constraints. Similarly to the above course scheduling experiments, as *variable ordering heuristic* we used minimum remaining values and degree for tie breaking, and we randomized it using the function in Lemma 1. No special *value ordering heuristic* was employed: the lexicographical order of the values was kept while iterating through them.

For each of these instances, we had 15 minutes to explore the search space. We recorded the best (lowest) solution costs found so far in Fig. 4.11 for several *conf* values. Approximately the same as in course scheduling, the lowest solution costs occur around $\text{conf} \approx 10$, which gives better results on average than the marginal *conf* values. This means that we achieve best results when the confidence to our heuristic is neither too high nor very low.

4.3.3 POPSSAMPLE during hard optimization

The *conf* parameter can refine any search method that adopts our heuristic framework. The POPSSAMPLE method goes a step further: it incorporates our heuristic *confidence semantics* into its search engine.

In order to solve the first university course timetabling instance (Fis0506-1 of Section 4.3.1), we invoked POPSSAMPLE for various PieceToCover and *conf* values and we plotted the best solution costs found in Figure 4.12. The third dimension is the *cost* of the solutions found: the lower the solution cost is, the more qualitative timetable is produced.

In the same graphs, we include some of the well-known search methods results, such as DFS, LDS, and Iterative Broadening, implemented in the same solver, with only their best solution cost depicted as a plane grid, in order to make comparisons easily.

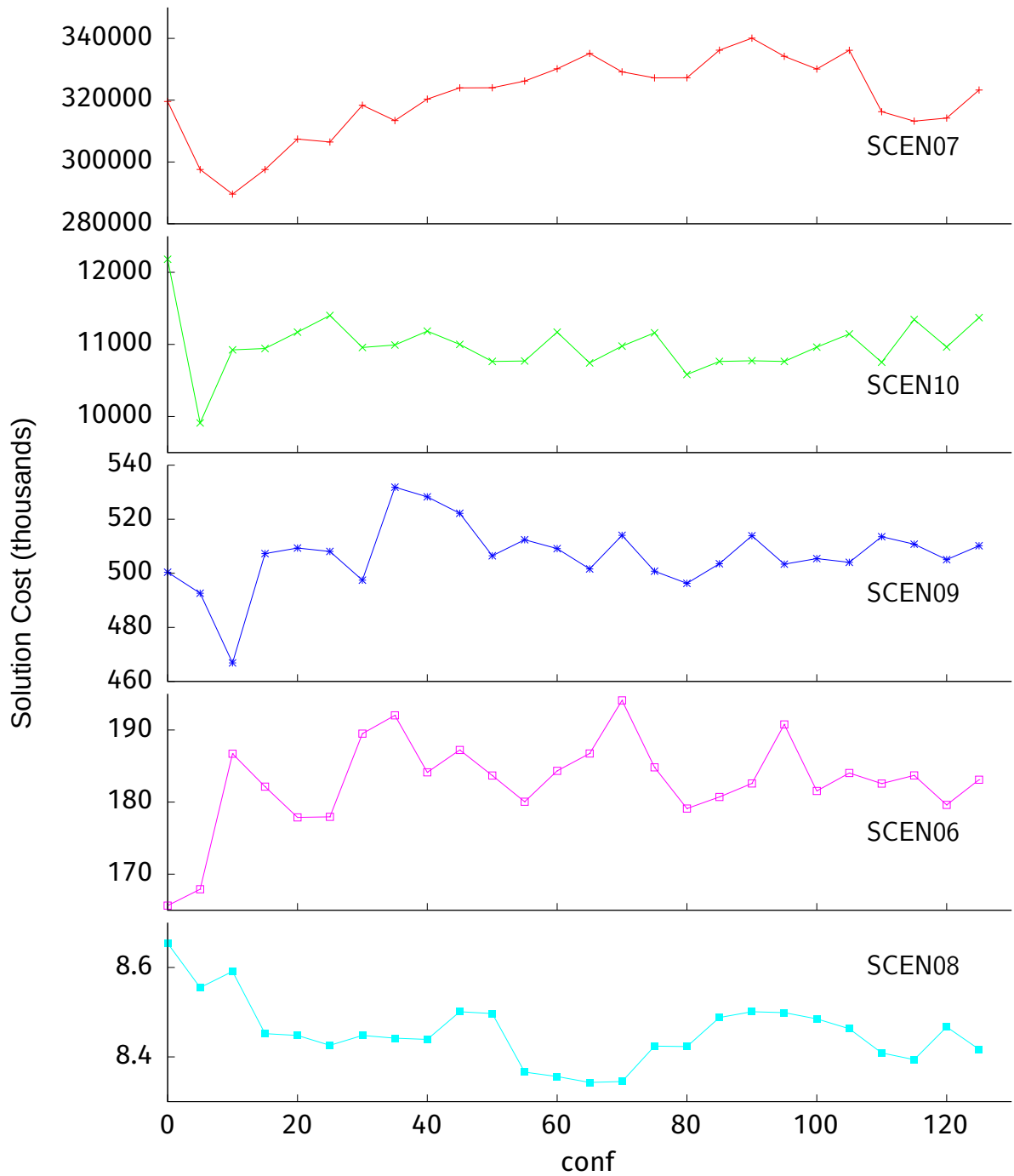


Figure 4.11: Unsatisfied soft constraints increase cost

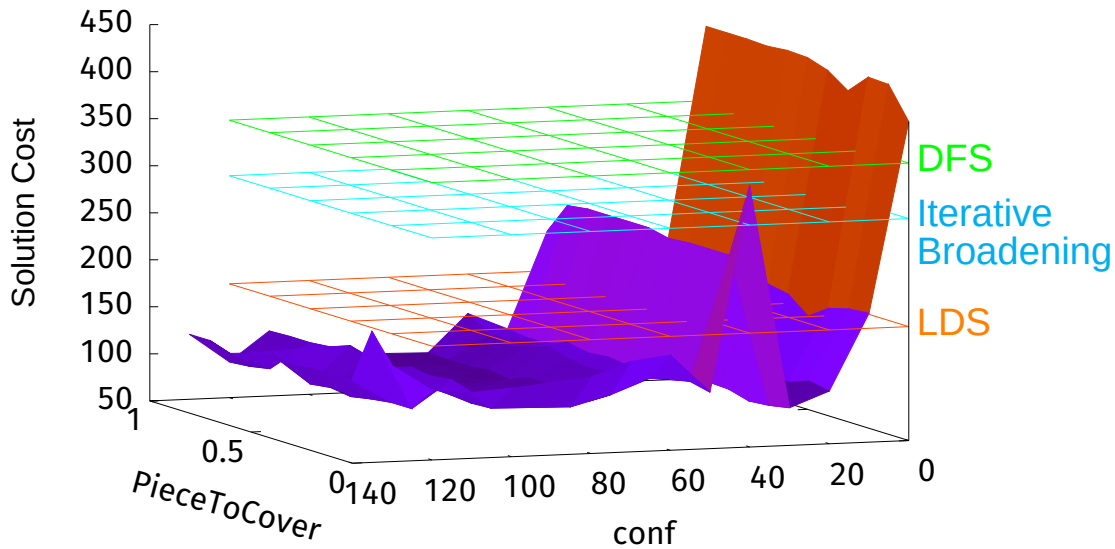


Figure 4.12: POPSAMPLE for the first ITC instance

4.3.4 PoPS vs. other search methods

In the above section, it was not easy to figure out the best PieceToCover and conf combination. In order to find it, we have to search by hand the lowest point in the three-dimensional PieceToCover vs. conf vs. Cost graph.

Now we employ PoPS to automatically seek for the best PieceToCover and conf combination while solving the fourteen course timetabling instances. For each instance, we have now just one solution as in Table 4.1.

As described in Section 4.2.4, PoPS uses several conf values and favors the most fruitful ones. We used five conf samples, i.e. 0, 25, 50, 75, and 100, by setting SamplesNum equal to 5. In this way, PoPS constructed solutions with lower costs than the other methods, except for the fifth instance, as illustrated in Table 4.1.

In this section, we used least constraining value as VALUESORDERHEURISTIC, and we randomized it using the function in Lemma 1. The time limit for all the methods was set to 15 minutes.

4.4 Conclusions

In this chapter, a new hybrid heuristic was defined. It has a confidence parameter to smoothly traverse from total determinism to total randomness. This heuristic can be adopted by any search method.

Additionally, this chapter introduced a new POPSAMPLE search method. This method efficiently exploits the proposed hybrid heuristic, as it increases the confidence parameter while descending to the leaves of a search tree.

Table 4.1: Solution costs for fourteen ITC instances

Instance	PoPS	LDS	DFS	It. Broad.
Fis0506-1	105	171	345	286
Ing0203-2	241	288	698	321
Ing0304-1	279	307	578	353
Ing0405-3	195	215	817	235
Let0405-1	655	627	X	X
Ing0506-1	307	311	812	342
Ing0607-2	282	283	1184	328
Ing0607-3	223	239	635	262
Ing0304-3	288	294	675	370
Ing0405-2	265	284	877	344
Fis0506-2	12	33	486	34
Let0506-2	713	783	1621	937
Ing0506-3	231	256	660	280
Ing0708-1	223	227	660	264

Finally, a new parameter-free PoPS search method is repeatedly calling `POPSAMPLE` with various values of its parameters and favors the values that produce the best results.

The efficiency of all the above has been illustrated in difficult real-world optimization CSPs.

5. CONSTRAINT PROGRAMMING MAPREDUCE'D

The world is one big data problem. There's a bit of arrogance in that, and a bit of truth as well.

Andrew McAfee, MIT

Search trees in Constraint Programming are a fertile ground for parallelization. However, it is difficult to propose a global parallelization schema, due to the great Constraint Satisfaction Problems (CSPs) variety and the plethora of the sequential search methods that are available to solve a CSP. In this chapter, we exploit a sequential search methods framework to make an arbitrary search method parallel by simulating its sequential execution. We record the visited search tree parts and then try to restore them on different Mappers-workers in a MapReduce installation [70].

5.1 Optimal search tree partitioning

Parallel search can benefit from splitting the search tree into equal parts. The most secure way to fairly split a search tree would be to traverse all of its nodes sequentially and record the elapsed real time when each node was visited. Then, we would divide the total time with the available workers number. Each time slice would be identified by two nodes: a start and an end node.

Take for example the tree in Fig. 5.1. Table 5.1 contains indicative times when the visit to each node was completed. It could be the real time in microseconds elapsed from the beginning of search. The visit to each node n_i (first row of the table) takes some time d_i (third row of the table). If we set 0 as the wall clock time when the visit to n_1 started, we start visiting node n_i when the wall clock time is

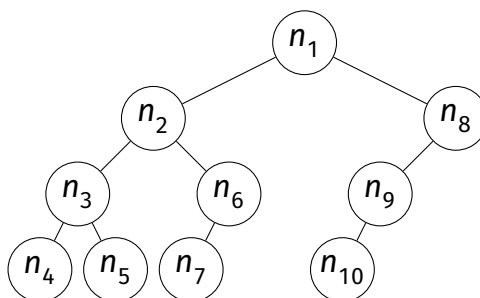


Figure 5.1: A search tree example

Table 5.1: The time when each node is visited sequentially

Node	n_1	n_2	n_3	n_4	n_5	n_6	n_7	n_8	n_9	n_{10}
Time visited	2	4	5	10	13	18	21	22	25	27
Duration	2	2	1	5	3	5	3	1	3	2

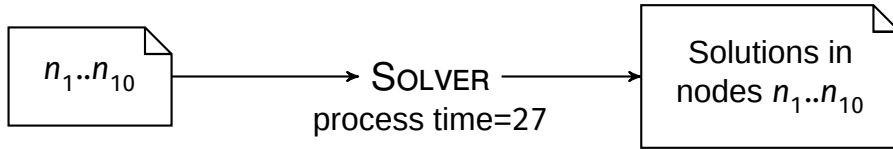


Figure 5.2: A sequential traversal finds all the solutions

equal to $\sum_{k=1}^{i-1} d_k$. The wall clock time needed to explore the whole search tree is $\sum_{k=1}^{10} d_k$. These aggregate times are illustrated in the second row of the table.

If we have 3 available workers to explore the above nodes, it would be better to divide the set of nodes into three almost equivalent parts. The desired duration for each part would be $27/3 = 9$ seconds. Hence, the ranges $n_1..n_4$ and $n_5..n_6$ and $n_7..n_{10}$ are almost equivalent, as they have the respective durations of 10, 8, and 9 seconds.

The aforementioned three parts of the search tree do not have any *topological* meaning; for example, they do not form three subtrees. The meaning of our partition is plainly *chronological*. For the first time, the search tree is not partitioned with a top-down approach; we focus directly on the factor of time.

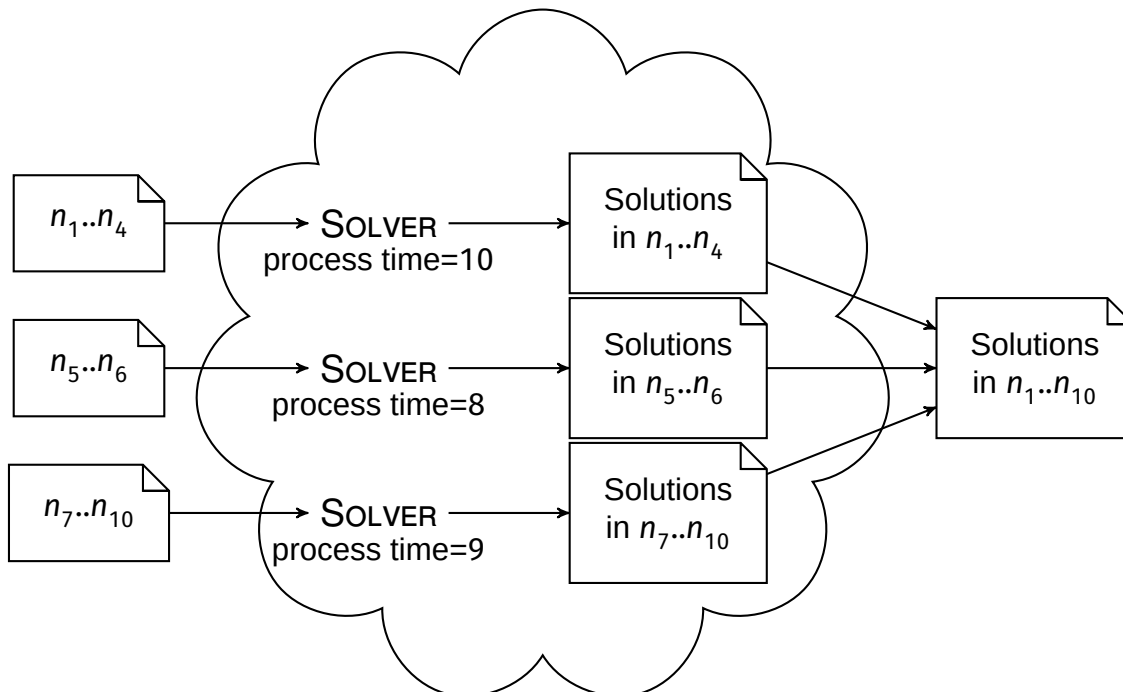


Figure 5.3: A parallel traversal of the tree by three Solvers

A sequential search method takes 27 time periods to traverse the whole search tree $n_1..n_{10}$, as in Fig. 5.2. If we had three workers to traverse the above three

splits $(n_1..n_4, n_5..n_6, n_7..n_{10})$ in parallel, in three different processors or cores, it would take only 10 periods, as in Fig. 5.3.

Note that the ranges $n_{start}..n_{end}$ that are the input for the Solvers are simply the IDs for two tree nodes and nothing more. In practice, the $n_{start}..n_{end}$ is a line in a text file, read by the Solver. But how can we encode a node into a text file?

5.2 Encoding a search tree node into an array of integers

The mechanism that searches for a solution in a Constraint Programming solver is a complex one as it has to support ad hoc customized search methods, as NAXOS SOLVER does [72]. For the sake of simplicity, we will focus on a single generic search method and not the whole search methods placeholder.

The following DFSVARORDHEUR recursive function is the backtracking search method Depth First Search, originally introduced in Fig. 2.4 as DFS. The difference is that DFSVARORDHEUR is more generic in the sense that it employs a dynamic variables ordering heuristic in contrast to DFS which has a static variables ordering. Furthermore, it aims to output every solution.

```

function DFSVARORDHEUR( $\ell$ )
  ▷ The method reached the search tree level  $\ell$ 
   $X \leftarrow \text{VARIABLESORDERHEURISTIC}(\mathcal{X})$ 
   $D_{X_{init}} \leftarrow D_X$ 
  for each  $v \in D_{X_{init}}$  do
     $D_X \leftarrow \{v\}$   ▷ Assign  $v$  to  $X$ 
    if no constraint is violated then
      ▷ Proceed to the next variable/level:
      if  $\ell = n$  then
        Print solution
      else
        DFSVARORDHEUR( $\ell + 1$ )
      end if
    end if
  end for
   $D_X \leftarrow D_{X_{init}}$ 
  return failure
end function

```

DFSVARORDHEUR aims, when called as DFSVARORDHEUR(ℓ) and $\ell - 1$ variables have already been evaluated, to assign a value to the ℓ -th variable (chosen by the heuristic) and then call itself to evaluate the rest variables. To solve a CSP, we initially call DFSVARORDHEUR(1). We are going to modify the above search method and give it the possibility to

- identify the node that is currently exploring and store it as a Sibling integer array,
- start exploring the search tree directly from a node described using a SiblingStart array, and

- terminate search when it visits a node identified as a SiblingEnd array.

The following DFSPART function integrates all the above features by adding the lines 4–16 to the original DFSVARORDHEUR function.

```

1: function DFSPART( $\ell$ , Sibling, SiblingStart, SiblingEnd)
2:    $X \leftarrow \text{VARIABLESORDERHEURISTIC}(\mathcal{X})$ 
3:    $D_{X_{\text{init}}} \leftarrow D_X$ 
4:   Sibling[ $\ell$ ]  $\leftarrow 0$ 
5:   for each  $v \in D_{X_{\text{init}}}$  do
6:     Sibling[ $\ell$ ]  $\leftarrow \text{Sibling}[\ell] + 1$ 
7:     if  $\ell \leq |\text{SiblingStart}|$  then
8:       if Sibling[ $\ell$ ] < SiblingStart[ $\ell$ ] then
9:         continue
10:      else if Sibling[ $\ell$ ] = SiblingStart[ $\ell$ ] then
11:         $\triangleright$  We found the starting sibling number!
12:         $\triangleright$  Nullify SiblingStart for current level:
13:        SiblingStart[ $\ell$ ]  $\leftarrow 0$ 
14:      end if
15:    end if
16:    if Sibling = SiblingEnd then
17:       $\triangleright$  All the items of the two arrays are equal
18:      terminate execution
19:    end if
20:     $D_X \leftarrow \{v\}$ 
21:    if no constraint is violated then
22:      if  $\ell = n$  then
23:        Print solution
24:      else
25:        DFSPART( $\ell + 1$ , Sibling, SiblingStart, SiblingEnd)
26:      end if
27:    end if
28:  end for
29:   $D_X \leftarrow D_{X_{\text{init}}}$ 
30:  return failure
31: end function
    
```

Figure 5.4 is an example of a search tree parsed by DFSVARORDHEUR. On top of each node n_i which is on level ℓ there is a number denoting Sibling[ℓ]: the serial number of the sibling that is implemented in DFSPART.

The path toward each n_i is unique and can be represented by the Sibling array in a unique way. For example, n_1 is represented by [1], n_2 is represented by [1, 1], n_3 by [1, 1, 1], n_4 by [1, 1, 2], n_5 by [1, 2], and so forth.

DFSPART is capable to partially explore the tree by restoring an n_{start} node and continuing the tree traversal until it meets a given n_{end} node. The two boundary nodes n_{start} and n_{end} are represented by the SiblingStart and SiblingEnd arguments of DFSPART.

Let us see the search tree and the DFSVARORDHEUR and DFSPART functions from the CSP solutions perspective. Let us suppose that the solutions of the CSP

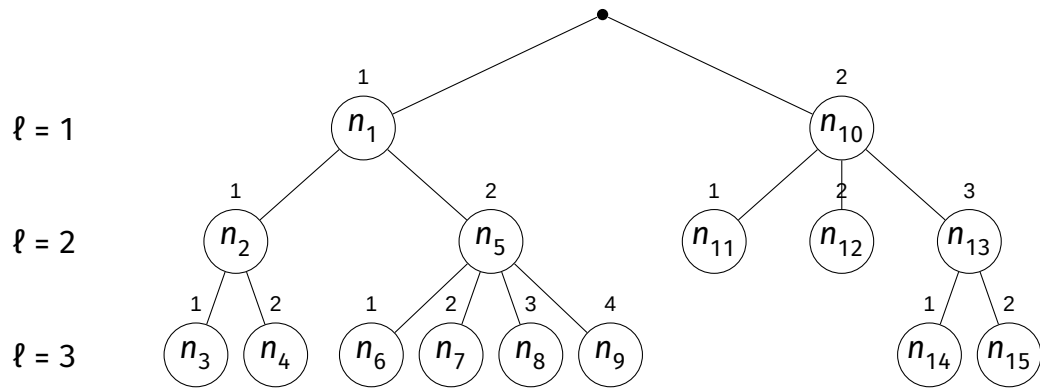


Figure 5.4: Tagging the nodes of a search tree

are depicted in the leaves-nodes $n_3, n_4, n_8, n_9,$ and n_{14} . This means that

- a $\text{DFSVARORDHEUR}(1)$ call should produce an output containing all these nodes encoded as
 - $[1, 1, 1]$
 - $[1, 1, 2]$
 - $[1, 2, 3]$
 - $[1, 2, 4]$
 - $[2, 3, 1]$
- $\text{DFS}\text{PART}(1, \text{Sibling}, n_1, n_7)$ will output only the solutions between n_1 and n_7 , i.e.
 - $[1, 1, 1]$
 - $[1, 1, 2]$
- $\text{DFS}\text{PART}(1, \text{Sibling}, n_7, n_{12})$ will output only the solutions between n_7 and n_{12} , i.e.
 - $[1, 2, 3]$
 - $[1, 2, 4]$
- $\text{DFS}\text{PART}(1, \text{Sibling}, n_{12}, n_{15})$ will output only the solution between n_{12} and n_{15} , i.e.
 - $[2, 3, 1]$

Therefore, in terms of solutions, $\text{DFSVARORDHEUR}(1)$ is equivalent to the above three independent and complementary DFSPART calls.

In other words, $\text{DFSVARORDHEUR}(1)$ iterates sequentially from the node n_1 to node n_{15} .

- $\text{DFS}\text{PART}(1, \text{Sibling}, n_1, n_7)$ is equivalent to the $\text{DFSVARORDHEUR}(1)$ execution as it iterates from n_1 (included) to n_7 (not included).
- $\text{DFS}\text{PART}(1, \text{Sibling}, n_7, n_{12})$ is equivalent to $\text{DFSVARORDHEUR}(1)$ execution as it iterates from n_7 to n_{12} .
- And $\text{DFS}\text{PART}(1, \text{Sibling}, n_{12}, n_{15})$ is equivalent to $\text{DFSVARORDHEUR}(1)$ execution as it iterates from n_{12} to n_{15} .

Finally, it is worth mentioning that the goal of partitioning the search tree is *not* to provide splits that contain equal number of solutions, but to create parts that can be explored in almost equal lengths of time.

Example 13. Let us suppose that we wish to explore in Fig. 5.4 only the nodes between n_7 (included) and n_{12} (not included). One should call `DFS_PART(1, Sibling, [1,2,2], [2,2])`. The last two arrays (`SiblingStart` and `SiblingEnd` arguments) represent n_7 and n_{12} respectively.

- In the first **for** iteration inside `DFS_PART`, we have `Sibling[1] = 1`. Thus, the condition statement `Sibling[l] = SiblingStart[l]` in line 10 is satisfied.
- Subsequently, `SiblingStart[1]` is set to zero in line 11. We have reached the correct Sibling, and therefore we will ignore `SiblingStart[l]` for $l = 1$ from now on.
- Execution continues, a value v is assigned to the constrained variable, and `DFS_PART(2, Sibling, SiblingStart, SiblingEnd)` is called. Now we have `Sibling[2] = 1`.
 - As `SiblingStart[2] = 2`, the condition in line 8 is satisfied, and `DFS_PART` directly continues to the second iteration of the loop, to reach the next sibling.
 - Now, `Sibling[2] = 2` which is equal to `SiblingStart[2]`. The equality condition in line 10 is met again, and `SiblingStart[2]` will be “nullified.”
 - We have reached n_5 in Fig. 5.4. A value v is assigned to the corresponding constrained variable. We are close to the starting node n_7 .
 - `DFS_PART(3, Sibling, SiblingStart, SiblingEnd)` is called.
 - * `Sibling[3] = 1` and `SiblingStart[3] = 2`. The condition in line 8 is satisfied. We continue to the second iteration of the loop, to reach the next sibling.
 - * `Sibling[3]` is now 2 which is equal to `SiblingStart[3]`. The equality condition in line 10 is met.
 - * We have eventually reached the starting node n_7 in Fig. 5.4 which is described by the `SiblingStart` array!
- `DFS_PART` proceeds with normal execution, without taking into consideration `SiblingStart` anymore.
- The search tree is regularly explored until we visit the ending node n_{12} which is described by `SiblingEnd`.

The time to restore the starting node is not important, as it is logarithmic in the search tree size.

5.3 Search tree nodes random sampling

The ideal would be to know a priori how to equally partition the search tree without the need to traverse it at first, as the time needed to traverse all the search tree nodes is equivalent to the sequential search time. Recall that our goal is not to reproduce sequential search, but simply to find some search space splits to explore them in parallel.

5.3.1 Pre-estimating a node's exploration time

Instead of exploring the whole search tree and then split it, we can traverse a representative proportion of it. Our partial traversal does not need to know anything a priori about the search tree. It does not need to know its size, height, etc. It will simply traverse a part of it.

This can be accomplished by overriding (“deleting”) a proportion of the search tree nodes and construct a table like Table 5.1, that will contain fewer nodes than the original one. This is a main contribution of this work: *To produce almost equal splits as the ones in Fig. 5.3 without having to keep every node in Fig. 5.1 and Table 5.1, but only the most representative.*

Sampling addresses two critical issues.

Issue 1. If a node n_i is overridden (passed by), how its time slice is replaced?

For example, if we override n_4 in Table 5.1, what would be the visit time for n_5 ? Overriding n_4 does not mean to completely ignore n_4 ; its corresponding duration should be added to the visit time of n_5 , because the new reduced table visit times should be as close to Table 5.1 real times as possible.

Issue 2. Deciding to override a node n_i leads us inevitably to override all of its offspring too.

E.g., the decision to override let us say n_6 in Fig. 5.1 is in fact a tough one: By overriding n_6 we override its offspring/descendant n_7 too.

But let us start sampling without considering the above issues initially.

Rule 1. Let R_i be a randomly generated real number, uniformly distributed in the range $[0, 1]$. Let n_i be a tree node without descendants and p the *simulation factor*, i.e. the minimum proportion of the nodes we want to override. Then, n_i is overridden if $R_i \leq p$.

This means that a node with no descendants is overridden with probability at least p . We say “at least,” because the precise probability to override n_i must additionally include the probability that one of n_i 's ancestors is overridden.

Now we should consider what happens if the node n_i has, let us say, d descendants. Note that the term “descendants” refers not only to the nodes (children) directly connected to n_i , but to all the nodes that belong to the sub-tree below n_i (the children of the children etc.). In this case, if we override n_i with probability p , this will override its descendants too. However, what we initially wanted was to override *each separate node* with probability p . Therefore, the probability to

override n_i and its d descendants should be:

$$\Pr[n_i \text{ overridden}] \cdot \Pr[1^{\text{st}} \text{ descendant overridden}] \cdots \Pr[d^{\text{th}} \text{ descendant overridden}]. \quad (5.1)$$

This is at least $p \cdot p^d = p^{1+d}$.

Rule 2. Let R_i be a randomly generated real number, uniformly distributed in $[0, 1]$. Let n_i be a tree node with d descendants and p the minimum proportion of the nodes we want to override. Then, n_i is overridden if $R_i \leq p^{1+d}$.

This is a simple workaround for Issue 2 above that also guarantees that the average proportion of the overridden nodes will be at least p .¹

5.3.2 Pre-estimating a node's descendants number

Rule 2 does not completely resolve Issue 2. When we are about to decide if a node will be overridden or not, with probability p^{1+d} , we should know the descendants number d . But this is not possible a priori, because we have not yet traversed the very node itself!

The solution is to make a *pre-estimation* of d , based on the previous history. In order to step forward, we make the following general assumption.

Assumption. Each node is expected to have a similar descendants number and a similar time duration to the other nodes that belong to the same tree level. In other words, the nodes that have equal distance from the root are expected to have similar descendants and duration.²

Take for example the lowest leaf nodes in Fig. 5.5. The node with the label t_4 is examined on whether is going to be simulated. At first, we need to pre-estimate its descendants number d . According to the above Assumption, the node with t_4 tag will have a similar d with the other nodes in the same level (t_1, t_2, t_3 tags). Each of these has zero descendants. Consequently, the average d is also $(0 + 0 + 0)/3 = 0$.

Hence, the node with t_4 tag will be overridden with probability $p^{1+0} = p$. And here comes Issue 1: If the node is indeed overridden, how its simulated duration t_4 will be computed?

The duration is computed exactly in the way that we computed d : as the average of the existing non-simulated nodes in the same level.

$$t_4 = \frac{\sum_{i=1}^3 t_i}{3}. \quad (5.2)$$

In fact, we put appropriate weights on the sum's terms, and we calculate the *weighted average duration*.

¹Again, we say "at least," because the precise probability to override a node must additionally include the probability that one of its ancestors is overridden.

²Besides, one node that is closer to the root is expected to have a larger lifetime than a node that is closer to the leaves. It can be demonstrated that the nodes of a specific level have a low standard deviation of their durations. Moreover, the multiple simulation (MapReduce) rounds described in a following section apply the simulation on smaller search tree parts, where the standard deviation is even smaller.

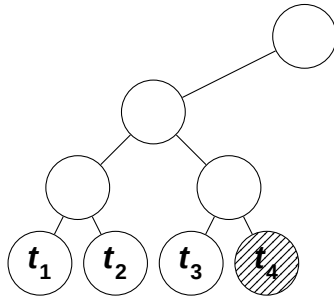
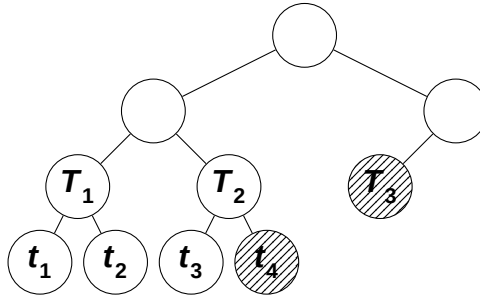


Figure 5.5: The fourth bottom node is going to be simulated


 Figure 5.6: T_3 is the simulation time for another node

Rule 3. If we want to override a node n_j , its *virtual time duration* is estimated as $t_j = \frac{\sum_{i=1}^{j-1} w_i t_i}{\sum_{i=1}^{j-1} w_i}$. The sum refers to the nodes n_i in the same level with n_j that have not been simulated themselves. The *descendants number* d_j of n_j can be pre-estimated exactly in the same way: $d_j = \frac{\sum_{i=1}^{j-1} w_i d_i}{\sum_{i=1}^{j-1} w_i}$.

Last but not least, we have to define the above w_i and ω_i so as to be flexible in cases such as the one in Fig. 5.6. In this figure we want to estimate the time T_3 (and, of course, the same applies to the corresponding descendants' number). Observe in the figure that T_2 is not as accurate as T_1 is, because T_2 includes the virtual duration t_4 that was computed in the previous paragraphs. Hence, T_2 is more virtual than T_1 and we have to reduce appropriately its weight.

Rule 4. The *weight for each time term* t_i in Rule 3 is calculated as $w_i = \frac{t_i - t_{\text{simulated}}}{t_i}$, where $t_{\text{simulated}}$ is the aggregate simulation time for the descendants of n_i . Quite similarly, the *weight for each descendants term* d_i is $\omega_i = \frac{d_i - d_{\text{simulated}}}{d_i}$, where $d_{\text{simulated}}$ is the sum of the d_i 's for the descendants of n_i that have been simulated and not visited in reality.

Note that if $t_{\text{simulated}}$ (or $d_{\text{simulated}}$) is zero, the weight w_i (or ω_i) becomes equal to 1. Experimental results about using the above w_i and ω_i formulas or simply setting them always to 1 are presented in the "Empirical Results" section.

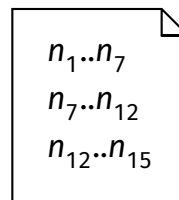
Finally, we *cannot* override the very first node n_i in each search tree level, as we cannot compute the duration t_i and descendants' number d_i based on the previous nodes in the same level.

5.4 The MapReduce input specification

In our MapReduce setup, a Mapper plays the role of a *worker* that has to explore a specific part of the search tree. The Reducer's role is trivial: an *identity* function that reproduces its input, i.e. the solutions provided by the Mappers.

Before reaching the Mappers, the MapReduce system normally gets an input text file as input. Then, it splits the text file into smaller ones, which are eventually distributed to the Mappers. The default and straightforward way to split a big text file is to create one file out of each line. In any case, splitting a large text file and distributing it to the Mappers is automatically done by the MapReduce system itself.

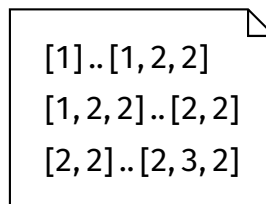
What is this input file in the first place? For Fig. 5.4 we may have the following text file as input.



```

n1..n7
n7..n12
n12..n15
    
```

As described in Section 5.2, the real content/encoding of the above is



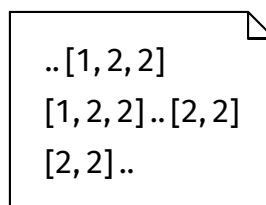
```

[1]..[1, 2, 2]
[1, 2, 2]..[2, 2]
[2, 2]..[2, 3, 2]
    
```

This is then split into three files: one file per line. Each of the three files “feeds” one Solver, as in Fig. 5.3.

The file containing the second line $[1, 2, 2]..[2, 2]$ will trigger $\text{DFS}_{\text{PART}}(1, \text{Sibling}, [1, 2, 2], [2, 2])$ as in Example 13.

Please note that the above text file can be furthermore simplified by omitting the very first and last nodes, as they are not necessary.



```

..[1, 2, 2]
[1, 2, 2]..[2, 2]
[2, 2]..
    
```

To sum up, from right to left, Reducers simply echo their input, and Mappers process one line which represents a part of the search tree. All the parts are stored in a big text file which is the input of the whole MapReduce system.

5.5 Slicing the search tree

Having specified the big data file format, it remains to be seen how this text file is initially created.

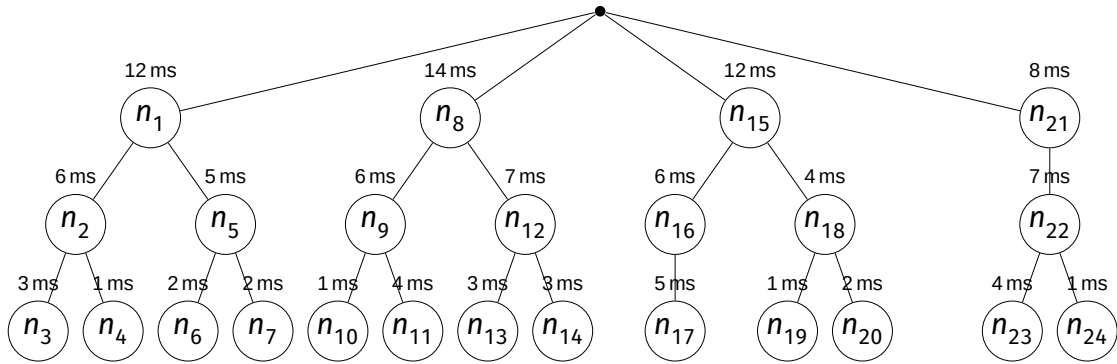


Figure 5.7: Needed time to explore each node and its descendants. The time of a parent node includes the time needed by its children.

5.5.1 Slicing the search tree by repeating sequential search

One simple way to generate the big text file (which represents the “big data” in the MapReduce terminology) is to execute DFSVARORDHEUR and record every X milliseconds the next search tree node to be visited using the aforementioned specified file format. Then, MapReduce would supply each line to a Mapper-Solver and expect that each of them would process the corresponding search tree part and output its solutions within X milliseconds.

Figure 5.7 presents a search tree with a tag above each node. Each tag represents the cumulative time (e.g. in milliseconds) needed to explore the node and its descendants (included).

Table 5.2 displays when each node in Fig. 5.7 is visited in a vertical timeline. The horizontal rules delimit the table into four parts that take about 12 milliseconds to be explored each and can be encoded to the following text file that could serve as a MapReduce input file.

```

n1..n8
n8..n15
n15..n21
n21..n24
    
```

For the sake of simplicity of the example, each time we make the transition from a parent node to a child (e.g. from n_1 to n_2 or from n_2 to n_3) we keep the clock unchanged. We change the clock when we have to move to a node of less or equal level, for example when we have to move from n_3 to n_4 or from n_4 to n_5 . In these cases, we find the previous node n_{prev} in the same level with the node we are currently visiting. We compute the current clock time by adding to the clock time of n_{prev} the time t_{prev} that we spent visiting it.

Unfortunately, this way of creating the big input file would last the same time as the whole sequential search, because we have to explore all the search tree nodes to construct it! We definitely need a faster way to generate this big text file in order to benefit from the MapReduce approach, end-to-end.

Table 5.2: A timeline while exploring the nodes in Fig. 5.7

clock	level 1	level 2	level 3
0	n_1		
		n_2	
3			n_3
6		n_5	n_4
8			n_6
			n_7
12	n_8		
		n_9	
13			n_{10}
18		n_{12}	n_{11}
			n_{13}
21			n_{14}
26	n_{15}		
		n_{16}	
32		n_{18}	n_{17}
33			n_{19}
			n_{20}
38	n_{21}		
		n_{22}	
42			n_{23}
46			n_{24}

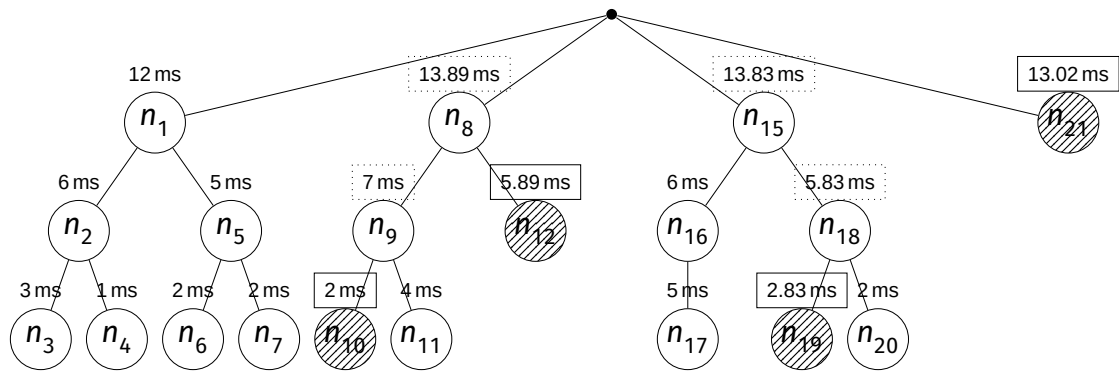


Figure 5.8: Estimation of the exploration time of the skipped (grayed-out) nodes

5.5.2 Slicing the search tree by mocking sequential search

The purpose of the big data generation is to output in a file only a tiny portion of the search tree nodes which delimit search tree parts that need an equal time to explore them. Hence, it is not necessary to visit all the search tree nodes in order to produce a small subset of them. As we already said, we can visit only a few of the nodes at the same depth and then skip the rest of them while adding to a virtual clock the estimated time we saved up while skipping these nodes.

Just like the approach where we interrupted search every X milliseconds and recorded the next node to be visited, now we interrupt search every X milliseconds of the virtual clock. Thus, we are able to speed up the big data generation, but we pay the price of not being accurate in having splits of exactly X milliseconds, as we use a virtual clock.

A simulation example

Figure 5.8 displays the search tree of Fig. 5.7 with some of its nodes overridden (simulated/grayed-out) and the subtrees below them deleted. Above each grayed-out node there is a box containing an *estimation* of the time that the exploration of this node would have taken. These estimations are used in order to proceed with sampling the search tree without exhausting all the nodes and without spending the time needed to visit all of them.

The nodes that are parents (or grandparents etc.) of the simulated search tree nodes have their exploration duration inside dotted boxes. Parent nodes include the time needed to explore their offspring, so the dotted boxes above them signify that part of the included time is an estimation, i.e. that the exploration time is partially real as some of the offspring have been simulated and overridden.

Similarly to the previous Table 5.2, Table 5.3 illustrates the times when each node of the search tree in Fig. 5.8 was visited. While this time is measured in milliseconds again, please note that the clock is *virtual*. This means that when we visit a grayed-out node, we make a leap in time: the virtual clock in the first column is updated without spending real time.

Table 5.3 contains a column with the probability p^{1+d} that the current node is overridden. For this example, we set $p = 0.1$. In practice, p will be normally set to something much greater than 0.1 in order to speed up simulation; this small value

Table 5.3: A timeline while sampling the search tree nodes

virtual clock	level 1	level 2	level 3	descendants pre-estimation	simulation probability	
0	n_1			—	—	
		n_2		—	—	
			n_3	—	—	
3			n_4	0	0.1	
6		n_5		2	0.001	
			n_6	0	0.1	
8			n_7	0	0.1	
12	n_8			6	0.0000001	
		n_9		2	0.001	
			n_{10}	0	0.1	simulated
14			n_{11}	0	0.1	
19		n_{12}		2	0.001	simulated
25.89	n_{15}			6	0.0000001	
		n_{16}		2	0.001	
			n_{17}	0	0.1	
31.89		n_{18}		1.75	0.0018	
			n_{19}	0	0.1	simulated
34.72			n_{20}	0	0.1	
39.72	n_{21}			5.63	0.00000023	simulated
52.74						

is used here plainly for illustrative purposes.

There is also a column with the pre-estimation of the descendants number d . For each row of the table, a random number in $[0, 1]$ is generated. If p^{1+d} is greater than the random number, then the current node is overridden/simulated, and this is noted in the last column. Everything depends on the random number and how big the probability is.

The very first node in each search tree level (n_1, n_2, n_3) was in purpose *not* simulated. They cannot be simulated as there are not any previous nodes in the same level that would allow to make a pre-estimation of how much time a node in this level would need to be explored and how many descendants it has.

Let us proceed with explaining Table 5.3 row by row. The visit durations (t_i values) are indicative.

- The virtual clock is initially set to 0.
- For the very first node of each search tree level, the sampling method is proceeding as usual: Each of the nodes n_1, n_2, n_3 is visited.
- For n_3 , the time $t_3 = 3$ and the descendants number $d_3 = 0$ is recorded.
- We are about to visit n_4 . From now on, we can use previous nodes to estimate the time that the next node would need and its number of descendants without having to visit it.
- Furthermore, from now on, we generate a random number in $[0, 1]$ for each node. If the random number is greater than p^{1+d} , we literally visit the node, else we simulate it.
- Based on the previous nodes of n_4 (only n_3) in the same search tree level, we pre-estimate d_4 to be 0. Therefore, the probability to simulate n_4 is $p_4 = 0.1^{1+0} = 0.1$.
- Let us say that the random number R_4 for this node is 0.6. As $p_4 < R_4$ we will not simulate this node.
- Thus, the search method is normally exploring n_4 and finally records $t_4 = 1$ and $d_4 = 0$.
- The search method then returns to n_2 for the last time. Before we return to n_1 , we record $d_2 = 2$ which is the total number of nodes under n_2 and $t_2 = 6$, which includes $t_3 + t_4$ plus 2 which is the time spent by n_2 itself.
- It is time to check if we will literally proceed to n_5 or simulate the visit. The (only) previous node in the same level had $d_2 = 2$ descendants, so we pre-estimate that d_5 will be 2 too. Therefore, the probability to simulate n_5 is $p_5 = p^{1+d} = 0.1^{1+2} = 0.001$.
- Let us say that the random number R_5 is 0.3. We have $p_5 < R_5$, so the node will not be simulated.

- In the same fashion, let us say that the leaves-nodes n_6 and n_7 will not be simulated too. We just traverse them and record the relevant times $t_6 = 2$, $t_7 = 2$, and, as we ascend back, $t_5 = t_6 + t_7 + 1 = 5$ and $t_1 = t_2 + t_5 + 1 = 12$.
- The corresponding d_i values are also recorded: $d_5 = 2$ and $d_1 = 6$, which is the number of all the nodes in the subtree under n_1 .
- Actually, we have completed the traversal of the leftmost subtree without simulating any of its nodes. Let us continue with n_8 .
- We pre-estimate the number of descendants d_8 based on the (only) previous node n_1 in the same level. Hence, d_8 is pre-estimated as 6. The probability to override/simulate n_8 is therefore $p_8 = p^{1+d} = 0.1^{1+6} = 0.0000001$. Let $R_8 = 0.35$. Again, the node n_8 will not be simulated as $p_8 < R_8$.
- We proceed to n_9 . We pre-estimate d_9 as the average $(d_2 + d_5)/2 = 2$. The probability to simulate n_9 is $p_9 = 0.1^{1+2} = 0.001$. For another time, let us suppose that $p_9 < R_9$: The node will not be simulated.
- Stepping one level deeper to n_{10} , the pre-estimation of d_{10} is 0, as the average of d_3 , d_4 , d_6 , and d_7 . Therefore, we have $p_{10} = 0.1$. Let $R_{10} = 0.05$. At last, this node will be simulated, as $p_{10} > R_{10}$. We will not spend any time here e.g. to propagate or validate constraints. We just record the estimation of time that a real visit to n_{10} would take as the average $t_{10} = (t_3 + t_4 + t_6 + t_7)/4 = 2$. We also set $d_{10} = 0$: the pre-estimation is permanently assigned to d_{10} .
- We are moving forward to n_{11} . The virtual clock counter is increased by $t_{10} = 2$, but this time is virtual: the actual time needed for the simulation was just the negligible time to compute an average value that is not comparable e.g. with the time needed to propagate constraints.
- n_{11} is eventually not simulated, we record $t_{11} = 2$ and $d_{11} = 0$, and we step back to n_9 .
- While we are leaving forever n_9 , we record $t_9 = t_{10} + t_{11} + 1 = 7$. We have also to record somehow that t_9 is “hybrid” time, in the sense that t_{10} is virtual time (an estimate) and t_{11} is a real visit time. For this purpose, we use a special weight defined in Rule 4 as

$$w_9 = \frac{t_9 - t_{\text{simulated}}}{t_9} = \frac{t_9 - t_{10}}{t_9} \approx 0.71.$$

The weights for all the other nodes on the same level are by default equal to 1. On the other hand, the weight of the simulated node n_{10} is equal to 0 as t_{10} is purely virtual.

- We also permanently record $d_9 = 2$ as the descendants number of n_9 .
- We proceed to n_{12} . Based on the previous nodes in the same level, we expect that it will have 2 descendants, as this is the average of d_2 , d_5 , and d_9 .

Hence, the probability to simulate it instead of visiting it is $p_{12} = p^{1+2} = 0.001$. It is a small probability, but let us suppose that the random number R_{12} is less than p_{12} .

- n_{12} is going to be simulated and, therefore, we should make estimations about t_{12} .

- The weighted average of the nodes in the same level is

$$t_{12} = \frac{w_2 t_2 + w_5 t_5 + w_9 t_9}{w_2 + w_5 + w_9} = \frac{1 \cdot 6 + 1 \cdot 5 + 0.71 \cdot 7}{1 + 1 + 0.71} \approx 5.89 \text{ ms.}$$

- Finally, it is estimated that $d_{12} = 2$.
- Back to n_8 , we compute $t_8 = t_9 + t_{12} + 1 = 13.89$ ms. The weight for t_8 is

$$w_8 = \frac{t_8 - t_{\text{simulated}}}{t_8} = \frac{t_8 - t_{10} - t_{12}}{t_8} \approx 0.43.$$

We have $d_8 = 2 + d_9 + d_{12} = 6$ descendants, as there are 2 direct descendants of n_8 plus the descendants of n_9 and n_{12} . The corresponding weight is

$$w_8 = \frac{d_8 - d_{\text{simulated}}}{d_8} = \frac{d_8 - d_{10} - d_{12}}{d_8} = \frac{6 - 0 - 2}{6} \approx 0.67.$$

- We move to n_{15} . To compute the simulation probability, we have to pre-estimate the descendants number. The pre-estimation is the weighted average of the descendants of the nodes in the same level

$$\frac{w_1 d_1 + w_8 d_8}{w_1 + w_8} = \frac{1 \cdot 6 + 0.67 \cdot 6}{1 + 0.67} = 6.$$

- The probability to simulate n_{15} is 0.1^{1+6} , but let us suppose that we will literally visit the node.

We continue to visit or simulate the rest of the nodes in the same fashion until the Figure 5.8 and Table 5.3 are completed. Finally, we are able to draw some horizontal rules in the table and split it into (virtual) parts of 12 ms.

Apparently, the durations of the partitions in Tables 5.2 and 5.3 do not coincide, as in the latter table some of the nodes have been skipped, and the measured time is less real and more “virtual” than the time in the former table. The virtual clock in Table 5.3 implies that we spent less time to construct the “big data” MapReduce input file.

5.5.3 How much does simulation cost?

The time needed to make a simulation is at most equal to the proportion $1 - p$ of the total time needed to explore the search tree.

In the above indicative example, the whole simulation process takes 90% of the total time needed to solve the CSP as $p = 0.1$. But this is clearly inefficient! In practice, we use much greater simulation probabilities, such as $p = 0.999$ as in the empirical results section.

5.5.4 Multiple MapReduce rounds

If the Mapper/worker reaches a given *timeout* (e.g. 60 seconds for a split that was pre-estimated to last 10 seconds) while trying to produce solutions, it stops. Then, the Mapper splits the remaining search tree part, and the Reducer records the splits to a common file that will be used as an input to the next MapReduce round. If in the current round no worker times out, then no splits are produced, and MapReduce has completed traversing the whole search tree.

Example 14. Let us say that a Mapper-worker gets the line $n_3..n_{10}$ as input. The worker has to traverse the search tree part between these two nodes. Let us also suppose that each line is supposed to be traversed within $X = 10$ ms.

Nevertheless, we have reached the timeout of 60 ms, and we are still in node n_6 . Unfortunately, we underestimated the duration of $n_3..n_{10}$.

In this case, we stop the traversal of n_6 , and we simulate again the traversal $n_6..n_{10}$ in order to produce new search tree parts (e.g. $n_6..n_8$ and $n_8..n_{10}$) and record them in a new input file for a new MapReduce round.

5.6 Empirical results with MapExplore

Based on the theory of the previous sections, we created MapExplore, a system that integrates Constraint Programming into the MapReduce framework. We integrated NAXOS SOLVER, our generic CSP solver written in C++ [67], inside Hadoop 2.7.1, a popular MapReduce environment. The source code for our evaluations is freely available.³

We installed Hadoop on eight Ubuntu Linux 14.04 virtual machines in the cloud [51, 52]. Each machine had eight 2 GHz CPU cores and 8 GB memory. The detailed hardware (e.g. CPU) specifications in the cloud are not available due to the so-called virtualization. One of the machines was selected to act as a coordinator (master) of the other seven machines (slaves). The master had a 60 GB disk and each slave a 40 GB disk. In our setup, the master machine also served as a slave machine, in order to save up as many CPU cores as possible.

5.6.1 Sequential vs. simulation time

Our NAXOS SOLVER is capable of solving any CSP. In these evaluations, we focused on N Queens and Number Partitioning and targeted to find all their solutions. The N Queens problem objective is to place N queens on a $N \times N$ chessboard so that no queen attacks any other. The objective of the N Number Partitioning problem is to split the set $S = \{1, 2, \dots, N\}$ into two disjoint sets S_1 and S_2 with equal cardinality, where N is even. Also, it should hold that $\sum_{i \in S_1} i = \sum_{j \in S_2} j$ and $\sum_{i \in S_1} i^2 = \sum_{j \in S_2} j^2$. Tables 5.4 and 5.5 display the time needed by a sequential search method to solve specific instances of these problems and find *all* the solutions of them.

³<http://di.uoa.gr/~pothitos/CPMR>

Table 5.4: Sequential time in seconds to solve N Queens

N	15	16	17
Sequential Search	1,212.7	7,902.5	57,910.9

Table 5.5: Sequential time in seconds for N Number Partitioning

N	40	44	48
Sequential Search	3,365.7	37,860.9	44,2013.4

5.6.2 MapExplore parallel/distributed execution time

In our introduced MapExplore system, the simulation incorporates a random factor: there is a probability whether a tree node will be visited or not (1‰ vs. 999‰, which imply a simulation probability $p = 0.999$). Thus, we repeated our experiments three times.

The figures that follow illustrate the *standard error* (SE) between the runs initiated with different random seeds. The standard error (SE) is defined as the ratio of the standard deviation (SD) of the recorded times to the square root of the number of samples, i.e. $SE = SD/\sqrt{n}$, where $n = 3$ repetitions. The standard error is depicted as an “I” on top of each measurement in the figures, just to get an idea of the possible variance between the samples. The standard error is not visible in most figures because it is small.

The simulation method provides a text file with many records in the format $n_{\text{start}}..n_{\text{end}}$. Each record is actually a part of the search space. The whole text file is the input for our MapReduce system. The file is automatically divided by Hadoop and each record is sent to a Mapper, which serves as a worker-SOLVER as in Fig. 5.3.

Each SOLVER instance traverses the search tree part that corresponds to its input record, outputs the solution in it, and then waits for another input record. MapReduce automatically gathers all the solutions found by all the Mappers into a single directory.

The whole wall clock time for our MapExplore process, from the search tree sampling to the solutions gathering is illustrated in Fig. 5.9 and 5.10. Each subfigure corresponds to a different CSP instance. The xx' axes depict the number of the Mappers used in our MapExplore system. The whole wall clock time includes the sampling phases and all the MapReduce rounds which were in general no more than five.

The Mappers number is very crucial as it is in fact the number of the workers/solvers employed. For most instances, the times are reduced while the Mappers number increases. This is something desirable as it seems that we can exploit to some degree every single available worker.

Nevertheless, the times for the Mappers number above 64 are reduced slightly or, especially for 15 Queens and $N=40$ Partitioning, get even bigger. The explanation for this behavior is that, in reality, we had only 64 CPU cores available. Even if we add more than 64 workers/solvers, these solvers have to share only

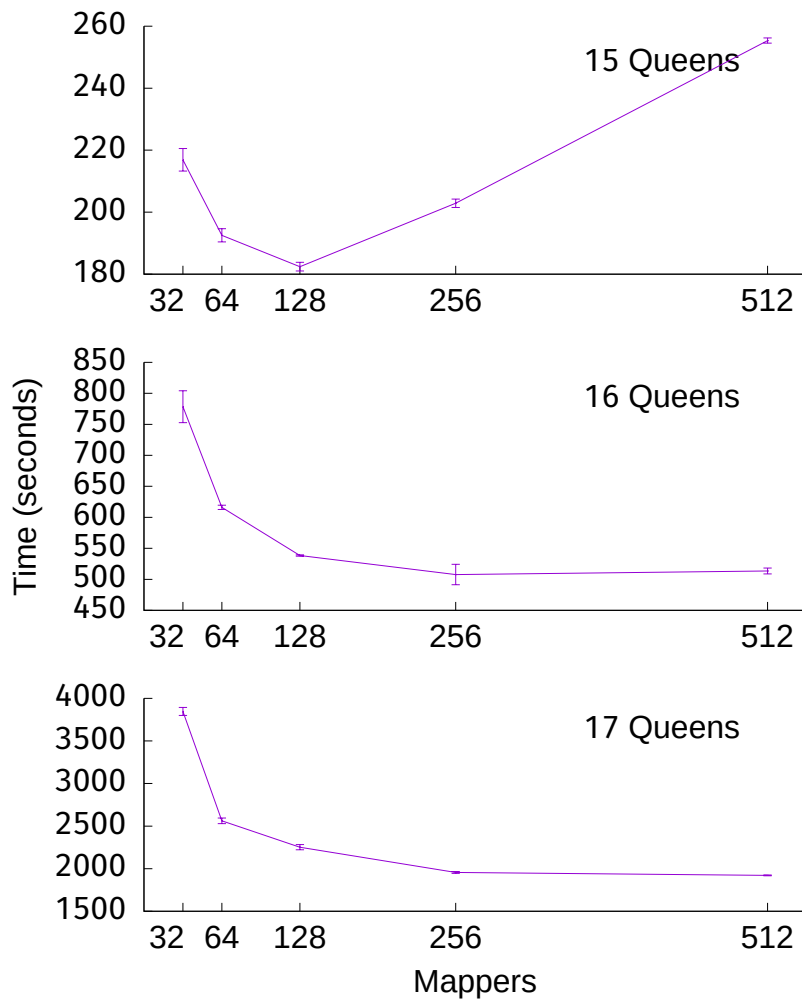


Figure 5.9: Time needed to get all the solutions of N Queens

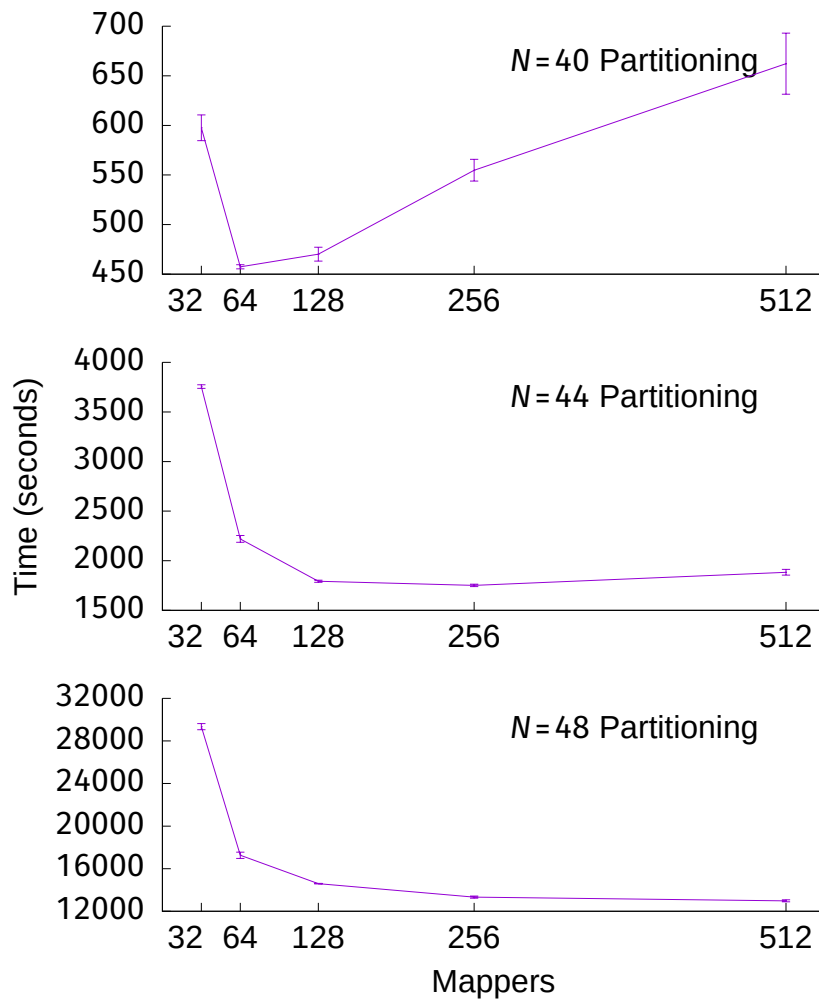


Figure 5.10: Time needed to get all the solutions of N Number Partitioning

64 cores; only 64 solvers can be active at the same time.

And why for smaller instances the times get worse as Mappers increase? The answer is that Hadoop and MapReduce in general are a framework for big inputs and outputs. Hadoop's overhead is apparent when creating more Mappers than needed.

Figures 5.11 and 5.12 display the corresponding *speedups* that MapExplore offers in relation to the sequential search process. The speedup is computed as the ratio of the sequential to the parallel execution time. As the instance gets bigger, we have bigger speedups, almost 35. Recall that MapReduce suits better to processes with big inputs/outputs.

Last but not least, in Fig. 5.13 one can see that if we use the weight w_i introduced in Rule 4 the overall time to solve N Queens instances is improved. For the experiments in Fig. 5.13 we used 128 Mappers.

5.7 Conclusions

Today, due to the availability of so many cores and virtual machines in the cloud, it is not enough to propose just efficient algorithms. These days, one has to create scalable algorithms and fairly distribute their execution to as many workers as possible. The state-of-the-art framework to achieve this is MapReduce, initially introduced by Google to process the whole Internet.

In this chapter, MapReduce was adopted to explore the huge search space of CSPs. MapReduce is designed to process big data files, so in this work the search space has been shredded into small parts, and the parts were encoded and assembled into a big text file. In related works, the search space is divided in a top-down manner. In this work, the search tree exploration was sampled, and for the first time the partitioning was done plainly in terms of time, without having to consider the search tree topology. Each small search tree part is a line in the big MapReduce input text file. Finally, it is the responsibility of MapReduce to distribute the lines of the big text file into the available solvers, to utilize all available resources, to keep a load balance between them, and to collect all the solutions found.

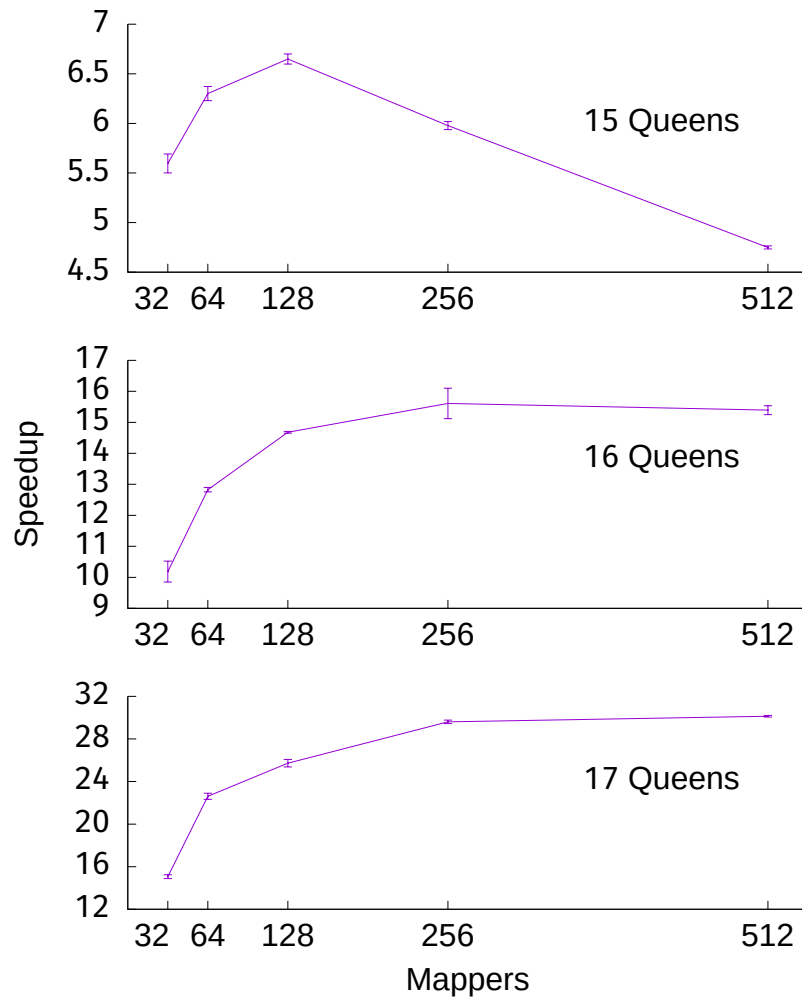


Figure 5.11: The speedup in relation to the sequential approach for the N Queens

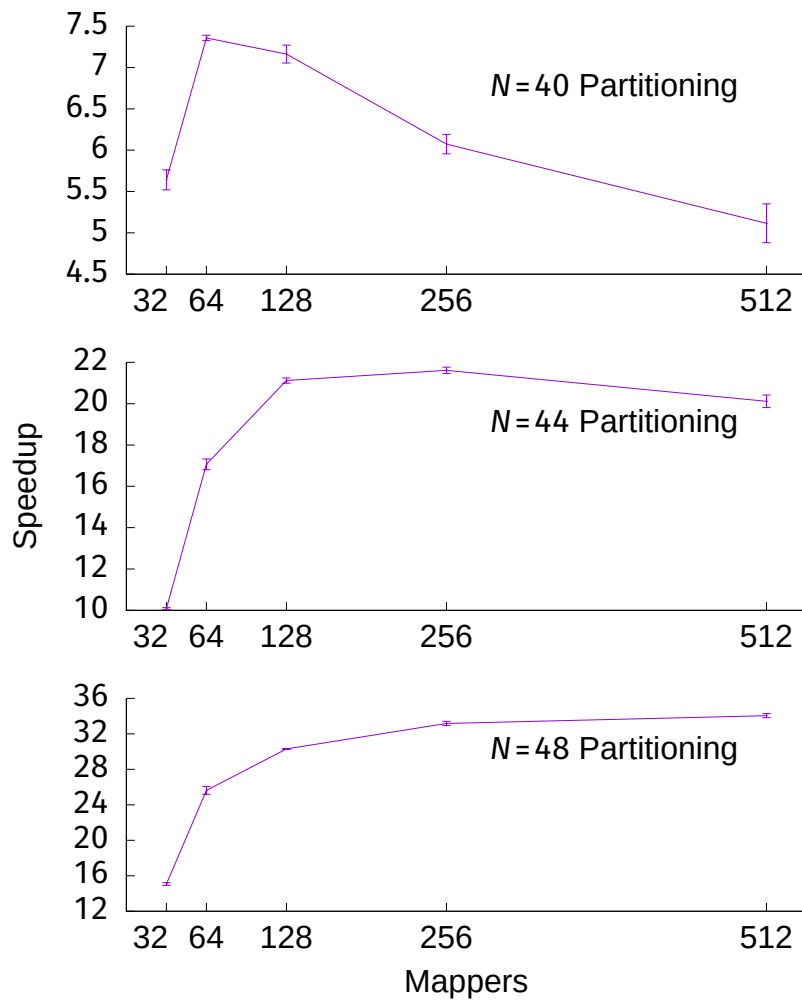


Figure 5.12: The speedup for the N Partitioning CSP

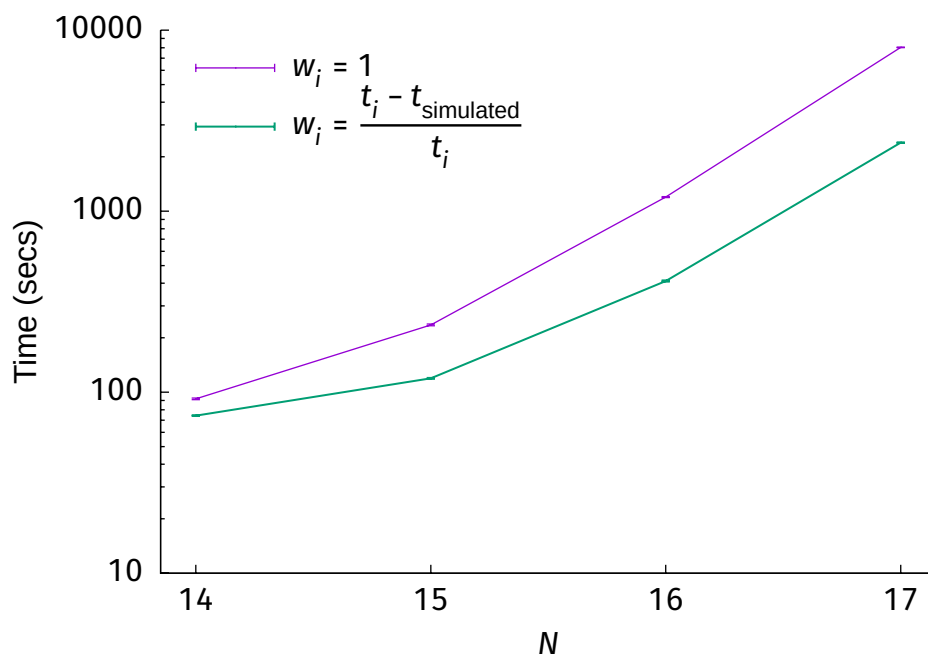


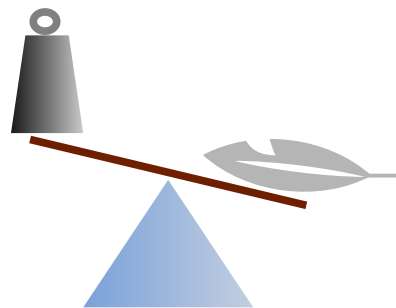
Figure 5.13: The weights in Rule 3 affect N Queens' times

6. THE REVENGE OF BOUNDS CONSISTENCY

My power is made perfect in weakness.

2 Corinthians 12:9

Arc consistency enforcement is an intelligent way to transform a Constraint Satisfaction Problem in order to reduce its search space. While current research focuses on stronger consistency levels than arc consistency, this chapter illustrates that in many practical cases the “weaker” bounds consistency can be used, making search more efficient. This paradox is theoretically explained for the first time [73].



We highlight *consistency enforcement* as an essential part of the solving process and we develop criteria that help Constraint Programming solvers select the fastest between arc consistency (AC) and bounds consistency (BC), without human intervention.

For the sake of simplicity and without loss of generality, in this chapter we will focus on binary CSPs. In binary CSPs, each constraint $C_{ij} = (S_{ij}, R_{ij})$ connects exactly two variables, i.e. $S_{ij} = \{X_i, X_j\}$. The R_{ij} set contains the valid combinations of the values of the two variables.

6.1 Consistency enforcement

Consistency is a particularly useful property in the road to solve a CSP. It implies that the values of the domains of each variable have a kind of *support* with respect to the CSP constraints. For the sake of readability, we repeat Definition 2 here as

Definition 10. An arc (X_i, X_j) is *arc consistent* iff for each $v_i \in D_i$ there exists a $v_j \in D_j$ with (v_i, v_j) not violating C_{ij} .

Example 15. Let X_1 and X_2 be two constrained variables with domains $D_1 = \{1, 2, 3\}$ and $D_2 = \{2, 3, 4, 5, 6, 7\}$. Let us assume that the constraint between the variables is $X_2 = 2X_1$.

(X_1, X_2) is arc consistent, as for each of the values 1, 2, 3 in D_1 , the corresponding values 2, 4, 6 belong to D_2 .

On the other hand, (X_2, X_1) is *not* arc consistent. To prove this, we need just one value from D_2 that does not have any support in D_1 . Indeed, for the value 3 in D_2 , there is not any v_1 in D_1 with $2v_1 = 3$.

If we want to make (X_2, X_1) arc consistent, we should remove the values 3, 5, 7 out of D_2 as they do not have any supports in D_1 .

This example also illustrates that consistency is not a symmetric property.

In order to check if an arc (X_i, X_j) is arc consistent, we have to iterate through all the values of D_j . The function that does this and removes the unsupported values from D_j is called **REVISE**. A faster yet looser alternative would be to check if the arc is *bounds consistent*.

Definition 11. An arc (X_i, X_j) is *bounds consistent* iff for the $\min D_i$ and $\max D_i$ values, there exist some $v_a, v_b \in D_j$ with $(\min D_i, v_a)$ and $(\max D_i, v_b)$ not violating C_{ij} .¹

In this case, **REVISE** has to check and update only the two bounds of D_j . But, in the worst case, when no support is found, it has to iterate through all D_j values too.

Example 16. Again, let X_1 and X_2 be two variables with $D_1 = \{1, 2, 3\}$, $D_2 = \{2, 3, 4, 5, 6, 7\}$, and $X_2 = 2X_1$.

(X_1, X_2) is bounds consistent, as for each of the bounds 1 and 3 in D_1 , the corresponding values $2 \cdot 1 = 2$ and $2 \cdot 3 = 6$ belong to D_2 .

Nevertheless, (X_2, X_1) is bounds inconsistent, as the upper bound 7 of D_2 has not any support in D_1 .

If we want to enforce bounds consistency to (X_2, X_1) , we should remove 7 out of D_2 . Note that only one removal is needed in the case of bounds consistency enforcement in contrast to the three removals needed by the arc consistency enforcement for the same domains in Example 15.

Lemma 4. Both arc and bounds consistency enforcement have equal time complexities in the worst case.

Proof. Time is measured by counting the number of elementary steps that each algorithm takes. We use the common *uniform unit system* in which every algorithm's operation takes the same constant time [96].

In order to compute the worst-case complexity of enforcing arc consistency, we repeat here the following procedure, already stated in Section 2.3.3.

¹Formally, Definition 11 is about the so-called *bounds(D)* consistency [10]. On the other hand, in the *bounds(Z)* consistency variant, v_a and v_b do not just belong to D_j but to its superset $[\min D_j .. \max D_j]$. Furthermore, the *range consistency bounds(R)* variant examines if every $v_i \in D_i$ (not just $\min D_i$ and $\max D_i$) has support in $[\min D_j .. \max D_j]$.

```

1: function REVISEAC( $X_i, X_j$ )
2:   domain_is_modified  $\leftarrow$  false
3:   for each  $v_i \in D_i$  do
4:     value_is_supported  $\leftarrow$  false
5:     for each  $v_j \in D_j$  do
6:       if  $(v_i, v_j) \in R_{ij}$ , with  $C_{ij} \in \mathcal{C}$  then
7:         value_is_supported  $\leftarrow$  true
8:         break
9:       end if
10:    end for
11:    if value_is_supported then
12:      continue
13:    else
14:      Remove  $v_i$  out of  $D_i$ 
15:      domain_is_modified  $\leftarrow$  true
16:    end if
17:  end for
18:  return domain_is_modified
19: end function

```

Let d be the maximum domain size. Then, line 3 performs at most d iterations. Each of the lines 2, 4, and 18 is 1 elementary operation.² The loop in line 5 performs at most d iterations. The statements inside this inner loop are at most 3 elementary operations. Finally, lines 11–16 consist at most 5 elementary operations.

Overall, we have at most $1 + d \cdot (1 + d \cdot 3 + 5) + 1$ elementary operations, which is $O(d^2)$.

Bounds consistency enforcement is a variation of the above.

```

function REVISEBC( $X_i, X_j$ )
  domain_is_modified  $\leftarrow$  false
  for each  $v_i \in D_i$  in ascending order do
    value_is_supported  $\leftarrow$  false
    for each  $v_j \in D_j$  do
      if  $(v_i, v_j) \in R_{ij}$ , with  $C_{ij} \in \mathcal{C}$  then
        value_is_supported  $\leftarrow$  true
        break
      end if
    end for
    if value_is_supported then
      break
    else
      Remove  $v_i$  out of  $D_i$ 
      domain_is_modified  $\leftarrow$  true
    end if

```

²We suppose that one elementary operation corresponds to one line of code execution. Russel and Norvig mention that “some measure that reflects the running time of the algorithm but is not tied to a particular compiler or computer [...] could be just the number of lines of code executed” [83].

```

end for
for each  $v_i \in D_i$  in descending order, with  $D_i \neq \emptyset$  and  $v_i > \min D_i$  do
  value_is_supported  $\leftarrow$  false
  for each  $v_j \in D_j$  do
    if  $(v_i, v_j) \in R_{ij}$ , with  $C_{ij} \in \mathcal{C}$  then
      value_is_supported  $\leftarrow$  true
      break
    end if
  end for
  if value_is_supported then
    break
  else
    Remove  $v_i$  out of  $D_i$ 
    domain_is_modified  $\leftarrow$  true
  end if
end for
return domain_is_modified
end function

```

Typically, REVISEBC is similar to REVISEAC, but contains two loops instead of one. The number of elementary steps inside each loop is still at most $(1 + d \cdot 3 + 5)$.

The number of iterations of the first loop plus the number of iterations of the second loop is at most d , because, in the worst case, the algorithm iterates through all the values of D_i . Each respective value of D_i is visited at most once.

Overall, similarly to REVISEAC, the number of elementary operations is again $1 + d \cdot (1 + d \cdot 3 + 5) + 1$ which is $O(d^2)$. \square

In a nutshell, enforcing arc or bounds consistency between a pair of constrained variables (X_i, X_j) takes the same time if X_i has not any support in X_j , which results in removing every value out of D_i . This is the worst case.

Nevertheless, in a better case, if REVISEBC finds a support, it stops the corresponding iteration through D_i values, while REVISEAC always iterates through all of them.

6.2 Our contribution and alternative approaches

From Constraint Programming early years, developers of solvers such as ILOG³ have observed empirically that there is a trade-off between arc and bounds consistency in terms of time and space, and bounds consistency is preferable in many cases [85]. More specifically, Barbara Smith quotes Jean-François Puget who mentioned

Solver is a compromise between efficiency and completeness... In the example [of constraint propagation of arithmetic constraints] the incompleteness comes from the fact that arithmetic expressions only

³<http://ilog.com>

propagate bounds. This is an example of the choice we made. Propagating holes in expressions [i.e. enforcing arc consistency] would require much more memory and time than the current implementation. From tests made on a very large set of examples, we found that the current compromise is by far better.

In alternative approaches to our work, in current constraint programming solvers, the choice between AC and BC is not justified theoretically but only empirically. In our work, apart from wide experimental results, we provide theoretical analysis for the AC vs. BC trade-off so as to predict when arc consistency becomes a bottleneck. We show that bounds consistency is usually more efficient when dealing with CSPs having large domains.

This could be thought of as a paradox, because AC and BC have equal worst-case complexities, and AC is stronger than BC, in the sense that it removes more inconsistent values out of the domains of constrained variables. This is true, but only when we study the constraint propagation algorithms isolated, independently of the search methods. In this work, we try to see the big picture: constraint propagation integrated into backtracking search methods. We compute the *overall* time complexity and focus on how it is affected by the choice between AC and BC.

In Section 6.3 we present the backbone of constructive search and the related mathematical notation. In Section 6.4 we compute the upper bounds of the complexities of search methods that traverse a path and maintain either AC or BC. In Section 6.5 we check in practice if the theoretically computed complexities can predict which methodology, AC or BC, fits better a given CSP. Finally, in Section 6.6 we introduce a bounds consistency variant that enforces consistency not to all (n) constrained variables but to a varying (k) number of them.

6.3 Constructive search

A *backtracking* approach involves a constructive search method that iterates through the constrained variables of a CSP: it assigns to the first variable a value and proceeds to the second variable, it assigns a value to it and, if the constraints are not violated, proceeds to the third variable and so on. *Backtracking* occurs if any of the constraints is violated: the current assignment is undone, and a different value is assigned to the variable. If all alternative values from the variable's domain are exhausted, we go to the previous variable and assign a different value to it and so on.

6.3.1 The typical backtracking search method

Figure 6.1 illustrates the recursive backtracking search method DFS (Depth First Search) originally introduced in Fig. 2.4. Each $\text{DFS}(\ell)$ call corresponds to the variable X_ℓ . In order to solve a CSP, we call $\text{DFS}(1)$, to begin with instantiating the first variable X_1 . This call attempts to assign to X_1 a value from D_1 ; hence, we may have at most d different attempts to assign a value to X_1 , where d is the maximum size of all the domains. Therefore, we have at most d subsequent calls of $\text{DFS}(2)$. Each $\text{DFS}(2)$ calls $\text{DFS}(3)$ and so on.

```

1: function DFS( $\ell$ )
    ▷ The method reached the search tree level  $\ell$ :
2:    $D'_\ell \leftarrow D_\ell$ 
3:   for each  $v \in D'_\ell$  do
4:      $D_\ell \leftarrow \{v\}$   ▷ Assign  $v$  to  $X_\ell$ 
5:     if no constraint is violated then
6:       ▷ Proceed to the next variable/level:
7:       if  $\ell = n$  then
8:         return success
9:       else if DFS( $\ell + 1$ ) = success then
10:        return success
11:      end if
12:    end for
13:     $D_\ell \leftarrow D'_\ell$ 
14:    return failure
15: end function

```

Figure 6.1: A typical search method

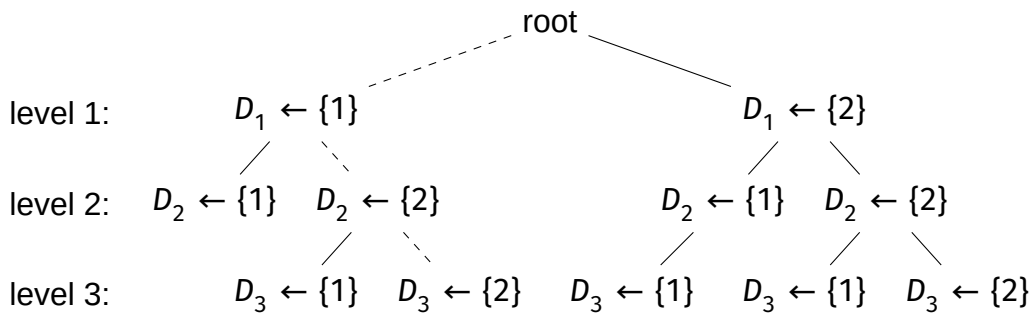


Figure 6.2: An incomplete binary search tree

For the sake of simplicity, a static variable ordering is kept while we assign values to the variables. Therefore, DFS(1) will assign a value to X_1 , DFS(2) will assign a value to X_2 , etc. Nevertheless, our computations are still valid even if we use another variable ordering heuristic.

This algorithm forms a *search tree*, as in Figure 6.2. The indicative CSP used in this figure contains three variables X_1, X_2, X_3 , with the corresponding domains $D_1 = D_2 = D_3 = \{1, 2\}$. Each level ℓ of the tree refers to a DFS(ℓ) call, and each node of the same level represents an iteration of its **for** loop. More specifically, each node is labeled with the assignment done in line 4.

We have at most d^n leaves representing the lowest level DFS(n) calls, where n is the number of the constrained variables.

Apart from DFS, there are many other constructive search methods [72]. In any case, DFS is the basis to describe most of them.

6.3.2 A search tree path

We denote as T_{path} the total time spent in the nodes that belong to the same path. A path begins from the root node and descends to a leaf node. The dotted line in Figure 6.2 is a path.

$T_{\text{path}}(\ell)$ is a part of T_{path} and denotes the time spent in a node of level ℓ while traversing a path.

In the rest of the chapter, the “AC” or “BC” exponents in the above symbols refer to the corresponding AC or BC methodology. For example, $T_{\text{path}}^{\text{AC}}(\ell)$ is the time spent in a node of level ℓ while maintaining AC.

6.3.3 Paths vs. trees

Throughout the rest of our theoretic computations, we measure the time spent in search tree *paths*, instead of focusing on the time spent while traversing all the paths of a complete search tree. This is done on purpose, just to simplify our computations.

After all, as it will be proved in the last theoretic section 6.4.4, if we manage to bound the time needed to traverse a search tree path, we are able to bound the time needed to traverse the whole search tree.

Therefore, we are going to compute respectively an upper bound for traversing a search tree path while maintaining AC or BC, and then multiply it by the maximum number of paths to get an upper bound for the whole search tree.

6.4 Maintaining consistency during search

Depth-first-search method complexity is exponential; we cannot actually decrease its complexity class, but it is possible to limit the number of nodes. In other words, we have to *prune* the tree to make search more efficient, and this can be done via enforcing and maintaining consistency.

6.4.1 Time complexity in a search tree node

Figure 6.3 illustrates a search method with an integrated consistency algorithm that can maintain either arc or bounds consistency. We break up the time spent by $\text{DFS_CONS}(\ell)$ when it is on the top of the call stack into four crucial parts.

- $T_{\text{prop}}(\ell)$ refers to the propagation algorithm in lines 2–5 and 9–10 respectively.
- $T_{\text{store}}(\ell)$ corresponds to line 6 of the algorithm and represents the time needed to store all the initial states of the domains.
- $T_{\text{restore}}(\ell)$ corresponds to line 18 and represents the time needed to restore all the domains. We claim that the time it takes to store the domains is equal to the time it takes to restore them, i.e. $T_{\text{store}} = T_{\text{restore}}$.

After all, storing the value of a variable requires transferring a specific number of bytes from one place of the memory to another. Re-storing the value back

```

1: function DFS_CONS( $\ell$ )
    ▷ Initially, enqueue all arcs and make them consistent:
2:   if  $\ell = 1$  then
3:      $Q \leftarrow \{(X_i, X_j) \mid C_{ij} \in \mathcal{C}\}$ 
4:     CONS( $Q$ )  ▷ See Figure 6.4
5:   end if
    ▷ Store a copy of the domains in  $\mathcal{D}$  for a future backtrack4
6:    $\{D'_1, \dots, D'_n\} \leftarrow \{D_1, \dots, D_n\}$ 
7:   for each  $v \in D'_\ell$  do
8:      $D_\ell \leftarrow \{v\}$ 
    ▷ Only the arcs toward  $X_\ell$  are enqueued:
9:      $Q \leftarrow \{(X_i, X_\ell) \mid C_{i\ell} \in \mathcal{C}\}$ 
10:    CONS( $Q$ )
11:    if not exists empty  $D_i \in \mathcal{D}$  then
    ▷ Proceed to the next level:
12:      if  $\ell = n$  then
13:        return success
14:      else if DFS_CONS( $\ell + 1$ ) = success then
15:        return success
16:      end if
17:    end if
    ▷ Restore the previous state of domains4
18:     $\{D_1, \dots, D_n\} \leftarrow \{D'_1, \dots, D'_n\}$ 
19:  end for
20:  return failure
21: end function

```

Figure 6.3: A search method that maintains consistency

```

function CONS( $Q$ )
  while  $Q \neq \emptyset$  do
    Remove an arc  $(X_i, X_j)$  out of  $Q$ 
    if REVISE( $X_i, X_j$ ) then
       $Q \leftarrow Q \cup \{(X_k, X_i) \mid C_{ki} \in \mathcal{C}, k \neq j\}$ 
    end if
  end while
end function

```

Figure 6.4: The core of a coarse-grained propagation algorithm (AC-3)

⁴The assignments of storing the domains or restoring them back do not necessarily mean to make a complete copy of the domain of every constrained variable. These two assignments imply the need to store/restore only the *modifications* to the domains done in the current search tree node.

to the variable (the original place of memory) involves the same number of bytes and, therefore, the same number of operations to transfer them back.

- T_{const} corresponds to lines 8 and 11–17. These statements take constant time.

In order to get the aggregate T_{path} time, we are going to compute the overall propagation and store-restore time for a search tree *path*, which is a route from the root of the tree ($\ell = 1$) to any of its leaves ($\ell = n$). This means that we will study the overall time of $\text{DFS_CONS}(1)$, $\text{DFS_CONS}(2)$, ..., $\text{DFS_CONS}(n)$ consecutive calls, each of them executing only one iteration of the **for** loop in line 7. The overall path time is at most

$$\begin{aligned}
 T_{\text{path}} &= \sum_{\ell=1}^n T_{\text{path}}(\ell) \\
 &= \sum_{\ell=1}^n T_{\text{prop}}(\ell) + \sum_{\ell=1}^n T_{\text{store}}(\ell) + \sum_{\ell=1}^n T_{\text{restore}}(\ell) + \sum_{\ell=1}^n T_{\text{const}} \\
 &= \sum_{\ell=1}^n T_{\text{prop}}(\ell) + 2 \cdot \sum_{\ell=1}^n T_{\text{store}}(\ell) + n \cdot T_{\text{const}}, \tag{6.1}
 \end{aligned}$$

as T_{const} remains the same for each ℓ , and, as previously explained, $T_{\text{store}} = T_{\text{restore}}$. This formula applies both to maintaining arc and bounds consistency algorithms. Nevertheless, according to the following table, there are some differentiations that are going to be elaborated on in the following sections.

Path time terms	$\sum_{\ell=1}^n T_{\text{prop}}(\ell)$	$2 \sum_{\ell=1}^n T_{\text{store}}(\ell)$	$n \cdot T_{\text{const}}$
Maintaining AC	$n^2 d^3$	$2nd$	$n \cdot \text{constant}$
Maintaining BC	— —	$2n^2$	— —
	Section 6.4.2	Section 6.4.3	

6.4.2 The constraint propagation aggregate complexity

Consistency enforcement algorithms are divided into two large categories: the *coarse-grained* and *fine-grained* algorithms [10]. The best algorithms from the two categories have been proven to have equal time complexities [12]. Therefore, without loss of generality, in order to study consistency enforcement as a whole, it suffices to simply focus on a typical coarse-grained algorithm, such as CONS in Figure 6.4.

CONS is initially called by DFS_CONS (Figure 6.3, lines 2–5) before actual search begins. The other propagation section (Figure 6.3, lines 9–10) inserts some more arcs into the Q and then invokes CONS again.

By replacing the two CONS calls in Figure 6.3 by its pseudocode in Figure 6.4 we are able to compute the overall time for the two propagation sections (lines 2–5 and 9–10) of DFS_CONS as the product of the number (E_{total}) of the inserted-removed arcs out of the Q and the time that REVISE takes.

We may have at most $E_{\text{total}} = n^2 \cdot d$ entry operations into the queue Q , where $n^2 \approx n(n - 1)$ denotes the maximum number of the arcs (X_i, X_j) with $X_i, X_j \in \mathcal{X}$ and $i \neq j$. After all, each specific arc (X_i, X_j) is initially inserted into the queue and also when a value is deleted out of D_j . Therefore, a specific arc is inserted at most $1 + d \approx d$ times into the queue, as a value cannot be deleted more than once while descending a search tree path.

In a search tree path, the domains gradually shrink, until they contain just one value in the last level or until a domain is “wiped out.” An arc (X_i, X_j) is enqueued when REVISE deletes a value from D_j , and also when X_j is assigned a value. An assignment is equivalent to deleting all the values in D_j , apart from one.

To conclude, we may have at most d deletions of values out of a domain, which can enqueue a specific arc. In sum, we may invoke at most d REVISE calls for a specific arc.

Following Section 6.1, a REVISE call takes approximately d^2 elementary steps. Overall, the propagation part of DFS_CONS will take approximately

$$\begin{aligned} \sum_{\ell=1}^n T_{\text{prop}}(\ell) &= E_{\text{total}} \cdot d^2 \\ &= n^2 d \cdot d^2 \\ &= n^2 d^3, \end{aligned} \tag{6.2}$$

which is the product of how many insertions we may have into the queue (E_{total}) and the REVISE function operations needed when an arc is popped out of the queue (d^2).

The same reasoning applies to faster—yet more complex—propagation algorithms [12]. The only difference is that these algorithms implement faster (but more memory-consuming) REVISE functions that still take the same time either for AC or BC.

Again, the important thing for the current theoretical analysis is that, in the worst case, the propagation time complexity remains the same, either while enforcing AC or BC. However, there are significant differences regarding the domains store and restore mechanism.

6.4.3 Backup and restore aggregate complexity

In the general case, constraint propagation cannot guide us directly to a solution. However, it can be a critical component of a backtracking search method: each assignment made is followed by consistency enforcement and each consistency enforcement is followed by an assignment.

If the constraints are violated, the last assignment is undone. This is a constant-time operation in a consistency-enforcement-free search method. But while a search method maintains consistency, the undo operation involves not only undoing an assignment, but also restoring the domains affected by the consistency enforcement after the assignment.

Why do we need to compute the “restore” time along with the “store” time? Theoretically, it would suffice only to *store* the modifications to the domains as we

descend a search tree path. But we need also to take into consideration the time needed to *restore* the domains back into their original state for two reasons.

1. No one can guarantee that the node we are currently visiting or the search tree path that we currently descend will ultimately guide us to a solution. In the worst case, we need to take into account the time needed to restore the domains into their previous state, before the current search tree node was visited. Then, we should try to visit another search tree node.
2. Even if the current search tree node does belong to a path that guides to a solution, we may need to find *all* the solutions and not only one. Therefore, in this case also we need to consider the time needed to undo the modifications done in the current search tree node.

Storing domains while maintaining arc consistency

As mentioned in Fig. 6.3, while descending a path, in each search tree node, we need to store (and then restore) all the modifications done to the constrained variables domains. By computing the total domain modifications number, we compute the minimum time needed to store them, while descending a search tree path.

AC enforcement may remove every value out of the domains of the n variables. The maximum domain size is d ; hence, we may have at most nd value removals. As we descend a search tree path (from $\ell = 1$ to n), each value can be only removed and not added back to a domain. Thus, the total values removed and stored for backtracking purposes in a single path is also bounded by

$$\sum_{\ell=1}^n T_{\text{store}}^{\text{AC}}(\ell) = nd, \tag{6.3}$$

which is the number of all the domain values in a CSP.

Example 17. Let us have four constrained variables X_1, X_2, X_3, X_4 with domains $D_1 = \{3, 4\}$, $D_2 = \{3, 5, 6\}$, $D_3 = \{0, 1, 2, 3, 4, 5\}$, and $D_4 = \{0, 2, 4, 6, 8, 10\}$. The constraints are $X_1 \neq X_2$, $X_1 \neq X_3$, $X_2 \neq X_3$, and $X_4 = 2X_3$.

The following table contains the changes that take place in the above domains, while searching for a solution to the problem.

Assignments	Updates in domains		
	D_2	D_3	D_4
$D_1 \leftarrow \{3\}$	3, 5, 6	0, 1, 2, 3, 4, 5	0, 2, 4, 6, 8, 10
$D_2 \leftarrow \{5\}$		0, 1, 2, 3, 4, 5	0, 2, 4, 6, 8, 10
$D_3 \leftarrow \{0\}$			0, 2, 4, 6, 8, 10
$D_4 \leftarrow \{0\}$			

Searching for a solution includes an assignment (first column) and enforcing consistency to the rest of the domains.

First, in the first row, we make the assignment $D_1 \leftarrow \{3\}$. As $X_1 \neq X_2$, we should remove 3 out of D_2 . Similarly, in the same row, we remove 3 out of D_3 as the

second constraint is $X_1 \neq X_3$. And as $X_4 = 2X_3$ and $2 \cdot 3 = 6$, we also remove 6 out of D_4 .

This was a practical example of arc consistency enforcement after an assignment takes place. We are still at the first level of the search tree.

As we proceed to the second row of the table, we make the assignment $D_2 \leftarrow \{5\}$. When we make an assignment, we proceed one level deeper into the search tree. Every assignment is followed by constraint propagation. In our case, we enforce arc consistency. As $X_2 \neq X_3$, we should remove 5 out of D_3 . And as $2 \cdot 5 = 10$, we remove 10 out of D_4 .

In the third row, we make the assignment $D_3 \leftarrow \{0\}$. The values 2, 4, and 8 are removed out of D_4 , as they do not have any support in D_3 anymore.

The last row is trivial, as we assign $\{0\}$, containing the only remaining value, to D_4 .

This was an example on how assignments interchange with constraint propagation during search. In the case of arc consistency constraint propagation, the domains eventually lose all their values. This is done gradually, while traversing the search tree levels. As we should be able to restore the domains in the state that they were in each search tree level, while descending a search tree path, we need to store every value of every domain (*nd values*).

Storing domains while maintaining bounds consistency

Again, while descending a search tree path, we need to know how many modifications will be done to the domains, in order to compute the minimum time needed to store (and then restore) them.

Bounds consistency can alter only the *bounds* of a domain. In order to store the previous bounds of a domain, we need 2 operations: to record the domain's lower bound and to record the domain's upper bound. At a search path node of level ℓ , the 2 operations can be repeated for every variable's domain; except for the variables that have been already instantiated, i.e. the variables having only one value in their domains.

These domains are excluded because there are not any other values in them that can be removed; if the last value is removed, we do not proceed, and we backtrack to a previous search tree level. In a search level ℓ , the instantiated variables are at least $\ell - 1$. Therefore, the uninstantiated variables are at most $n - \ell + 1$. The overall time needed to store the initial domains in a search tree node in level ℓ is

$$T_{\text{store}}^{\text{BC}}(\ell) = 2(n - \ell + 1), \quad (6.4)$$

which is the product of the two operations needed to store the two bounds of a variable, and the number of uninstantiated variables.

For all the nodes of the search tree path it holds

$$\begin{aligned}
 \sum_{\ell=1}^n T_{\text{store}}^{\text{BC}}(\ell) &= \sum_{\ell=1}^n (2(n - \ell + 1)) \\
 &= 2 \sum_{\ell=1}^n n - 2 \sum_{\ell=1}^n \ell + 2 \sum_{\ell=1}^n 1 \\
 &= 2n^2 - 2 \frac{n(n+1)}{2} + 2n \\
 &= n(n+1) \approx n^2.
 \end{aligned} \tag{6.5}$$

Example 18. Let us consider the same constraint satisfaction problem as in the previous Example 17.

The following table depicts the state of the domains during search. Each row corresponds to a search tree level. The table is different from the one in Example 17, in the sense that it does not contain every value of every domain, but only their *bounds*.

Assignments	Updates in domains bounds					
	D_2		D_3		D_4	
	min	max	min	max	min	max
$D_1 \leftarrow \{3\}$	5	6				
$D_2 \leftarrow \{5\}$			0	4	0	8
$D_3 \leftarrow \{0\}$					0	0
$D_4 \leftarrow \{0\}$						

Again, the assignments interchange with constraint propagation. After the first assignment $D_1 \leftarrow \{3\}$, we have to enforce *bounds* consistency. This means that the minimum and maximum values of every domain should have supports to the other constrained variables. If a bound of a domain does not have any support, it is trimmed.

The initial minimum value of D_2 is 3. But as $X_1 \neq X_2$ and $D_1 = \{3\}$, this value is not supported. Therefore, it should be removed out of D_2 and 5 becomes its new minimum value.

Then, we make the assignment $D_2 \leftarrow \{5\}$. As it holds that $X_2 \neq X_3$, the upper bound of D_3 which is 5, is not supported anymore. That is why in the second row of the table, $\max D_3$ has been trimmed to 4. Subsequently, due to the $X_4 = 2X_3$ constraint and as the maximum value 10 of D_4 is not supported now, we delete it, and 8 becomes the new $\max D_4$.

In the third row, we assign $\{0\}$ to D_3 . In this case, $\max D_4$ should become 0 too, as this is the only supported value through the $X_4 = 2X_3$ constraint.

This example illustrates that, in every search tree level, we need to store only the bounds of the domains of the unassigned constrained variables, which is the meaning of the above equation (6.4).

6.4.4 Will arc or bounds consistency be faster?

The answer to this question is unknown before we actually start and finish solving a given arbitrary CSP. There is not any exact mathematical form to know a

priori how much time each search methodology will take either while maintaining AC or BC.

Nevertheless, we can bound the time needed by these search methodologies using the above equations to compute the respective *path* times $T_{\text{path}}^{\text{AC}}$ and $T_{\text{path}}^{\text{BC}}$. These two path times allow us not to compute the exact times for AC and BC (that will be simply denoted as TIME_{AC} and TIME_{BC} in the rest of the chapter) but at least to get the respective upper bounds $\text{TIME}_{\text{AC BOUND}}$ and $\text{TIME}_{\text{BC BOUND}}$.

Proposition 1. If $n < d$, then $\text{TIME}_{\text{AC BOUND}} > \text{TIME}_{\text{BC BOUND}}$, else if $n > d$, then $\text{TIME}_{\text{AC BOUND}} < \text{TIME}_{\text{BC BOUND}}$.

Proof. TIME_{AC} and TIME_{BC} is bounded by T_{path} if we multiply it by the maximum number of paths. The maximum number of paths is equal to the maximum number of leaves d^n . Therefore,

$$\text{TIME}_{\text{AC BOUND}} = d^n \cdot T_{\text{path}}^{\text{AC}}, \quad (6.6)$$

$$\text{TIME}_{\text{BC BOUND}} = d^n \cdot T_{\text{path}}^{\text{BC}}. \quad (6.7)$$

By combining (6.1) and (6.2) we get

$$T_{\text{path}} = n^2 d^3 + 2 \sum_{\ell=1}^n T_{\text{store}}(\ell) + n \cdot T_{\text{const}}. \quad (6.8)$$

We specialize the above equation for AC and BC via (6.3) and (6.5).

$$T_{\text{path}}^{\text{AC}} = n^2 d^3 + 2nd + n \cdot T_{\text{const}}, \quad (6.9)$$

$$T_{\text{path}}^{\text{BC}} = n^2 d^3 + 2n^2 + n \cdot T_{\text{const}}, \quad (6.10)$$

which leads to Proposition 1, because

$$\begin{aligned} & n < d \\ \Leftrightarrow & 2n \cdot n < 2n \cdot d \\ \Leftrightarrow & n^2 d^3 + 2n^2 + n T_{\text{const}} < n^2 d^3 + 2nd + n T_{\text{const}} \\ \Leftrightarrow & T_{\text{path}}^{\text{BC}} < T_{\text{path}}^{\text{AC}} \\ \Leftrightarrow & d^n T_{\text{path}}^{\text{BC}} < d^n T_{\text{path}}^{\text{AC}} \\ \Leftrightarrow & \text{TIME}_{\text{BC BOUND}} < \text{TIME}_{\text{AC BOUND}}. \quad \square \end{aligned}$$

6.4.5 Discussion

To compute the overall complexity of exploring a search tree and maintaining arc/bounds consistency, we considered the worst case, i.e. that all the possible leaves of the search trees will be visited. One may argue that this is a paradox, as the purpose of maintaining consistency is to prune as many leaves and paths in the search tree as possible and never visit all of them.

This is true, but we considered visiting the whole search tree, as this facilitated the mathematical formulas, and, after all, we just needed an *upper bound* for the time needed to maintain arc/bounds consistency during search. Therefore,

the times used plainly for constraint propagation (T_{prop}) either for arc or bounds consistency were considered equal.

In contrast to the time needed to *propagate* the changes in the constrained variable domains, we focused on the times that a backtracking mechanism needs to *store* these changes, and we found significant differentiations while maintaining arc or bounds consistency.

We believe, furthermore, that the time needed to store the changes of the domains has an immediate relationship to the *memory* needed by the two propagation methodologies themselves. Conclusively, we believe that even when the propagation times T_{prop} remain the same both for arc and bounds consistency, there is still a differentiation in the *memory* needed to maintain each consistency level (same as the above differentiation between $T_{\text{store}}^{\text{AC}}$ and $T_{\text{store}}^{\text{BC}}$) that will unavoidably affect the respective propagation *times* too in practice.

6.5 Empirical evaluations

All the above theory was inspired by observations while solving artificial and real-life constraint satisfaction problems. To test the theoretical results of this work in practice, we consider all standard CSP instances taken from the First XCSP3 Constraint Mini-Solver Competition [32]. The specific instances used in the *mini-solver* track are available under the respective link in the competition site.⁵

Tables 6.1, 6.2, and 6.3 display raw experimental results, while Figure 6.5 depicts them graphically. But, before going through all these empirical results, let us describe how one can reproduce them.

6.5.1 Methodology

In order to make comparisons, we had to employ two different solvers: one that maintains arc consistency (AC) and another that maintains bounds consistency (BC). Therefore, we took the open source NAXOS SOLVER [67] and created its AC and BC variants.

Note that the original NAXOS SOLVER implements several consistency levels for various constraints. Consequently, we created two sets of patches, one that implements pure arc consistency and another for pure bounds consistency for every constraint employed. All patches are freely available.⁶

Similarly to the theory of this work, we considered only binary constraints (that apply between two constrained variables) to simplify consistency enforcement. Therefore, we binarized the global constraints (that apply to more than two variables) that exist in some CSP instances by substituting them by groups of equivalent binary constraints.

Finally, it is worth noting that, in order to be more accurate, the illustrated CSP parameters n and d (number of constrained variables and maximum domain cardinality in the CSP) are not taken directly from the CSP definition; they are

⁵<http://www.cril.univ-artois.fr/XCSP17>

⁶<https://github.com/pothitos/ACvsBC-Solver-Patches>

Table 6.1: CSP attributes and solution times while maintaining AC and BC – Part I

CSP	n	d	TIME _{AC}	TIME _{BC}
aim-50-2-0-unsat-2	50	2	0.71	0.65
AllInterval-007	25	13	0.07	0.07
AllInterval-012	45	23	0.14	0.13
AllInterval-016	61	31	0.20	0.19
AllInterval-035	137	69	0.65	0.91
AllInterval-050	197	99	1.42	5.27
AllInterval-080	317	159	6.65	203.52
bdd-15-21-2-2713-79-08	21	2	41.39	40.37
bdd-15-21-2-2713-79-16	21	2	2326.85	X
bqwh-15-106-35_X2	106	6	0.43	4.53
bqwh-15-106-36_X2	106	6	0.21	1.46
bqwh-18-141-09_X2	141	6	7.99	597.87
bqwh-18-141-31_X2	141	7	0.33	828.42
bqwh-18-141-83_X2	141	6	5.18	837.63
color_X2	500	5	68.21	X
ColouredQueens-03	9	3	0.01	0.01
composed-25-01-25-3	33	10	0.09	0.04
composed-25-01-25-4	33	10	0.09	X
composed-25-10-20-5	105	10	0.31	1481.76
composed-75-01-25-6	83	10	0.22	X
cril-5_X2	42	81	55.06	X
Crossword-m1c-lex-h1501	225	26	11.34	X
Crossword-m1c-ogd-h2310	529	26	40.89	83.52
Crossword-m1c-uk-vg-4-8	32	26	11.39	14.81
Crossword-m1c-words-p20	81	26	0.65	0.58
driverlogw-01c	71	4	0.02	0.02
driverlogw-02c	301	8	110.01	X
driverlogw-04c	272	11	3.03	50.28
driverlogw-08c	408	11	263.16	X
driverlogw-08cc	408	11	254.62	X
Dubois-021	63	2	149.08	140.53
Dubois-022	66	2	303.95	291.51
ehi-85-297-30	297	7	84.50	0.33
ehi-85-297-98	297	7	0.32	0.34
ehi-90-315-13	315	7	0.27	0.30
ehi-90-315-37	315	7	0.34	0.33
geometric-50-20-d4-75-03	50	20	0.50	0.51
geometric-50-20-d4-75-46	50	20	13.79	X
geometric-50-20-d4-75-54	50	20	0.30	0.66
jnh-012	100	2	0.17	0.12
jnh-213	100	2	0.12	0.08
jnh-302	100	2	0.09	0.13
Kakuro-easy-015-sumdiff	194	9	0.07	0.05
Kakuro-easy-079-sumdiff	344	9	0.13	0.11

Table 6.2: CSP attributes and solution times while maintaining AC and BC – Part II

CSP	n	d	TIME _{AC}	TIME _{BC}
Kakuro-easy-084-ext	240	9	0.48	0.40
Kakuro-easy-109-ext	256	9	0.97	1.15
Kakuro-easy-150-ext	256	9	0.79	0.78
Kakuro-easy-164-sumdiff	344	9	0.30	0.30
Kakuro-hard-179-sumdiff	996	9	2.02	666.57
Kakuro-medium-016-ext	140	9	0.15	0.15
Kakuro-medium-020-ext	140	9	0.09	0.09
Kakuro-medium-055-sumdiff	234	9	0.09	0.05
Kakuro-medium-162-ext	256	9	13.81	14.25
Langford-3-05	25	11	0.12	0.12
Langford-4-04	28	9	0.13	0.15
Langford-4-05	35	14	0.17	0.17
MagicHexagon-02-0000	18	7	0.14	0.09
MagicSquare-3-sum	17	9	0.02	0.01
MagicSquare-3-table	9	9	0.01	0.01
MagicSquare-4-table	16	16	0.24	0.11
MagicSquare-5-table	25	25	139.94	1627.44
MarketSplit-03	151	100	1149.24	264.75
MarketSplit-05	153	99	674.29	309.90
MarketSplit-07	152	100	595.51	148.30
MarketSplit-08	154	100	292.73	115.03
MarketSplit-09	152	100	570.38	270.14
MarketSplit-10	151	100	243.32	142.11
MultiKnapsack-1-03	235	2536	14.62	0.87
MultiKnapsack-1-5_X2	239	4106	X	95.23
MultiKnapsack-2-16	274	1181	X	76.94
MultiKnapsack-2-21	342	1361	X	46.15
MultiKnapsack-2-22	342	1501	X	163.88
MultiKnapsack-2-41	136	1126	73.21	2.90
MultiKnapsack-2-48	180	1126	811.88	43.36
Nonogram-018-table	576	2	3.14	2.98
Nonogram-035-table	576	2	2.48	2.49
Nonogram-096-table	576	2	5.79	5.73
Nonogram-168-table	400	1	1.20	1.04
Nonogram-177-table	1024	2	2.57	2.69
Nonogram-180-table	1024	2	32.95	34.15
Pb-queen-0974553	1137	39	25.70	3.47
pigeonsPlus-07-05	42	7	17.76	8.76
pigeonsPlus-08-04	40	8	52.36	28.48
pigeonsPlus-09-03	36	9	239.93	143.16
Primes-10-20-3-3	213	784	10.79	0.04
Primes-10-60-3-3	444	784	46.51	659.36
Primes-15-20-2-5	219	2116	168.98	0.16

Table 6.3: CSP attributes and solution times while maintaining AC and BC - Part III

CSP	n	d	TIME _{AC}	TIME _{BC}
Primes-20-40-2-1	241	3574	71.23	0.05
PropStress-0020	293	24	1297.90	1.30
qwh-10-57-7_X2	100	5	0.11	0.10
rand-2-23-23-253-131-0	23	23	673.35	X
rand-2-23-23-253-131-1	23	23	663.37	X
rand-2-30-15-306-230f-09	30	15	8.51	65.62
rand-2-40-11-414-020-23	40	11	46.38	328.02
rand-2-40-11-414-020-35	40	11	4.67	68.40
rand-5-12-12-200-12442-38	12	12	642.24	846.58
rand-5-12-12-200-t95-3	12	12	701.10	803.31
rand-5-2X-05c-15	12	12	558.94	1837.32
Renault	101	42	3.87	3.66
Renault-medium-pos	148	20	0.27	0.30
Renault-megane-pos	99	42	3.05	3.53
Renault-mgd	101	42	3.74	3.48
Renault-small	139	16	0.08	0.06
Renault-souffleuse	32	12	0.01	0.02
RenaultMod-09	111	42	740.11	1032.59
Sat-flat200-06-dual	2237	4	470.44	261.04
Sat-flat200-14-dual	2237	4	1.34	179.40
Sat-flat200-32-dual	2237	4	130.75	13.55
Sat-flat200-55-dual	2237	4	146.12	1147.39
Sat-flat200-65-sum	6911	3	127.33	117.12
Sat-flat200-67-dual	2237	4	138.31	688.10
Sat-flat200-80-dual	2237	4	X	1574.29
SchurrLemma-mod-012-9	12	9	6.88	39.40
SchurrLemma-mod-015-9	15	9	9.45	37.19
SchurrLemma-mod-020-9	20	9	11.85	42.86
SchurrLemma-mod-030-9	30	9	17.74	58.30
SchurrLemma-mod-050-9	50	9	33.20	100.52
SchurrLemma-mod-100-9	100	9	116.89	301.21
Subisomorphism-A-15	180	200	2.61	13.11
Subisomorphism-g07-g39	20	1	0.06	0.05
Subisomorphism-g08-g31	30	100	4.83	7.06
Subisomorphism-g10-g35	41	120	0.05	0.04
Subisomorphism-si2-b09-m200-02	40	200	0.30	0.21
Subisomorphism-si6-b03-m800-07	480	800	1.23	1.43
TravellingSalesman-20-076_X2	61	70	50.54	1407.10
TravellingSalesman-20-142_X2	61	115	1640.18	X
TravellingSalesman-25-003_X2	76	62	190.14	X
TravellingSalesman-25-066_X2	76	62	28.76	224.92
TravellingSalesman-4-20-001-a4_X2	61	52	60.84	448.12
TravellingSalesman-4-20-727-a4_X2	61	74	708.48	X

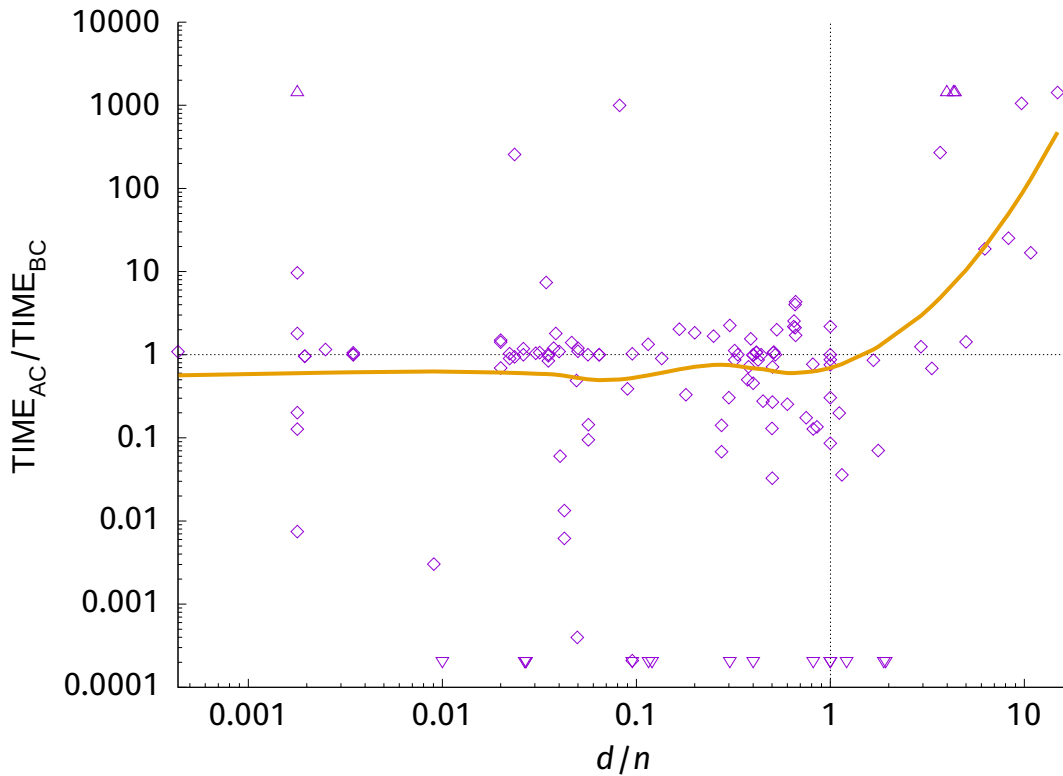


Figure 6.5: The time needed to solve the CSPs while maintaining AC divided to the time spent while maintaining BC

reported by the solver itself. Consequently, n is reported only after the binarization of the constraints has been completed, possibly by adding more constrained variables.

Also, we enforce bounds consistency for the first time, before displaying the maximum domain cardinality d . This means for example that if we have two constrained variables X_1 and X_2 , with $X_1 \leq X_2$ and $D_1 = \{1, 2, \dots, 100\}$ and $D_2 = \{25, 26, \dots, 50\}$, the maximum cardinality will not be computed as 100. Bounds consistency will be enforced first, and D_1 will be limited to $\{1, 2, \dots, 50\}$. The maximum domain cardinality d will be eventually displayed as 50. In this way we “normalize” redundant domains.

6.5.2 Execution

In order to construct Tables 6.1 and 6.2 with the experimental results, we follow the above methodology and display n and d for each CSP instance. If n is greater than d , we display it bold, else d is displayed bold. In theory, when d is greater than n , we expect that maintaining bounds consistency is more efficient than maintaining arc consistency.

In the above tables, if the corresponding TIME_{AC} for a CSP is bold it means that it is less than TIME_{BC} . Otherwise, TIME_{BC} is bold, which means that maintaining bounds consistency is more efficient than arc consistency in this CSP.

Using the above methodology, we created two separate solvers, one that maintains arc consistency and one that maintains bounds consistency. Each of

them was assigned to solve the First XCSP3 Constraint Mini-Solver Competition CSPs [32]. Each CSP instance has to be solved within 40 minutes according to the competition standards. If a solver cannot solve an instance within this time frame, it is marked with an “X” in the table. Otherwise, the elapsed time in seconds is written. Please note that only the CSP instances that were solved at least from one solver are displayed in the table.

We executed the experiments in an Ubuntu Linux 18.04 virtual machine with 8 virtual CPUs and 8 GB of memory.

6.5.3 Visualization

In order to make comparisons more easily, we depicted graphically the ratio $\text{TIME}_{AC}/\text{TIME}_{BC}$ versus d/n in Figure 6.5 using the \diamond symbol.

When the AC solver does not produce a solution, we have an undefined TIME_{AC} denoted as “X” in the table. In the figure, the corresponding point is depicted with a \triangle symbol. This represents a very high $\text{TIME}_{AC}/\text{TIME}_{BC}$ ratio, which means that maintaining BC is much more efficient than AC in this case.

On the other hand, when TIME_{BC} is “X,” the ratio $\text{TIME}_{AC}/\text{TIME}_{BC}$ is depicted with a ∇ symbol. This denotes a very low ratio, which means that AC is much more efficient than BC in this case.

It may be obvious that the above \triangle and ∇ points do not correspond to real values. They are used in the margins of Figure 6.5 to represent marginal ratios, as described above.

As the \diamond points in the figure are somehow sparse, the results become more intuitive if we draw a smooth curve between them. Therefore, the curve in Figure 6.5 has been derived by the LOESS method [23, 97] and is representative of the \diamond points.

In rough lines, LOESS is used to unify scattered points along the plot by drawing a smooth curve that passes between them. The advantage of this method is that it does not require a parameter or function of any form to fit a model to the data. The only input is the data themselves.

In our case, we made LOESS method ignore the marginal \triangle and ∇ points because they do not depict real values.

6.5.4 Observations

In Figure 6.5 we compare the times for solving a CSP instance via maintaining AC and BC. A first conclusion is that BC can be better than AC for many instances. This is an important observation, as, due to the fact that AC enforces a stronger consistency level than BC, and both AC and BC have equal worst-case complexities (Lemma 4), there is the misconception that AC is always better than BC.

However, the conclusion about the occasional superiority of BC over AC has no practical use, if we do not know when it happens. We have to find the appropriate conditions to know a priori if a CSP instance will be solved faster by maintaining AC or BC.

In theory (Proposition 1) the relation between n and d defines the relation between the upper limits of TIME_{AC} and TIME_{BC} . To put it simply, the d/n ratio

affects the $\text{TIME}_{AC}/\text{TIME}_{BC}$ ratio, and this is evident in practice in Figure 6.5: On average, $\text{TIME}_{AC}/\text{TIME}_{BC} < 1$ if $d/n < 1$ and $\text{TIME}_{AC}/\text{TIME}_{BC} > 1$ if $d/n > 1$. This becomes clearer if we observe the smooth curve constructed by the LOESS method, which represents the “average” of the \diamond points [23].

Of course, there is some deviation between our theoretic expectations and the observed results. This is due to the fact that in theory we studied the worst case of complete search trees for both maintaining AC and BC, while in practice the two methodologies may produce incomplete search trees that are different between them.

Regarding the \triangle points (that represent the cases when only the maintaining BC method found a solution while maintaining AC did not find one) they are apparently more on the right side, i.e. when $d/n > 1$. On the other hand, the ∇ points are gathered mostly on the left side of Figure 6.5. This means that for $d/n < 1$, the maintaining BC methodology is usually not only less efficient than AC, but it may produce no solution for a CSP, while AC is able to solve it.

6.6 The new k -bounds-consistency variant

We have shown that, under certain conditions, maintaining bounds consistency can be more efficient than maintaining arc consistency. What about going one step further? Can we loosen bounds consistency itself—by enforcing it not to all arcs but to a subset of them—and produce even more efficient results?

We are going to propose a looser consistency type, which enforces bounds consistency only to the variables with domain sizes less than or equal to k [74].

6.6.1 Theoretical analysis

Definition 12. The arc/constraint (X, Y) connecting the variables X and Y is k -bounds-consistent, iff (X, Y) is bounds-consistent or $|D_X| > k$.

Example 19. Let X, Y be constrained variables with the corresponding domains $D_X = \{5, 6, 8\}$ and $D_Y = \{1, 2, 3\}$, and it holds $X = Y + 5$. The constraint is *not* 5-bounds-consistent, because $|D_X| \leq 5$ and there is not any support in D_Y for $5 \in D_X$. However, we do have 2-bounds-consistency, as $|D_X| > 2$.

Lemma 5. k -bounds-consistency enforcement on an arc (X, Y) requires at most $O(kd)$ steps.

Proof. The revision of an arc/constraint includes the *check* for support values and the consistency *enforcement*. To check for the consistency, we need $O(2 \cdot |D_Y|)$ steps, as for each one of the two D_X bounds, we try to find a support value $y \in D_Y$.

But we must also consider what happens when a D_X bound is found inconsistent. In this case we should *enforce* bounds-consistency by removing the inconsistent bound out of D_X and by repeating the above *check* for the new bound. We may have $O(|D_X|)$ removals.

As a result, the overall revision complexity is $O(|D_X|) \cdot O(2|D_Y|) = O(|D_X| \cdot |D_Y|)$.

Nevertheless, remember that k -bounds-consistency is enforced only when $|D_X| \leq k$. As a consequence, k -bounds-consistency has a $O(k \cdot d)$ worst case cost. \square

As k grows and approaches infinity, which is as a matter of fact equivalent to d , k -bounds-consistency becomes evidently identical to bounds-consistency. For low k values k -bounds-consistency approximates a simple constraint check.

Maintaining 1-bounds-consistency during search is identical with a plain backtracking method, and no constraint propagation is done. In this case 1-bounds-consistency degenerates into a way to check if a constraint is satisfied: We search to find a support value for the unique bound/value of X ; if no support is found, the unique value is removed, and an ultimate inconsistency signal is broadcast. This particular consistency type may also appear in “lazy” propagation schemas, e.g. in local search contexts [68].

6.6.2 Empirical results

Following the experiments in Section 4.3.1, we enforce our new consistency level while solving the fourteen real-world datasets of the International Timetabling Competition (ITC) track for universities [58]. All the source code is freely available.⁷ The experiments were conducted on an HP computer with an Intel dual-core E6750 processor at 2.66 GHz and 2 GB of memory, running Ubuntu Linux 8.04. In accordance with ITC standards, we have only 334 seconds in this machine in order to find a solution.

The lightweight consistency proposed seems in theory to ease the burden of the necessary revisions. But is it competitive in demanding problems such as real-life course timetabling, in relation to other consistency levels?

Figure 6.6 displays the corresponding costs of the solutions found for each one of the fourteen datasets. It is obvious that for each one of them there is a specific k , for which maintaining k -bounds-consistency methodology gives the best results. For $k = 1$, the methodology actually uses no constraint propagation; it is a plain backtracking method, so the results are poor. On the other hand, while k approximates infinity, i.e. while k -bounds-consistency approaches plain bounds-consistency, the results are not so poor, but are apparently worse than using the k value, which usually lies around 25. We may consider this value something like a “golden mean” but only for this type of CSP instances.

Conclusively, for very small k values we found low quality solutions, i.e. with high cost. On the contrary, as k increases above the “golden mean,” the solution quality remains almost the same.

⁷<http://di.uoa.gr/~pothitos/ictai2012>

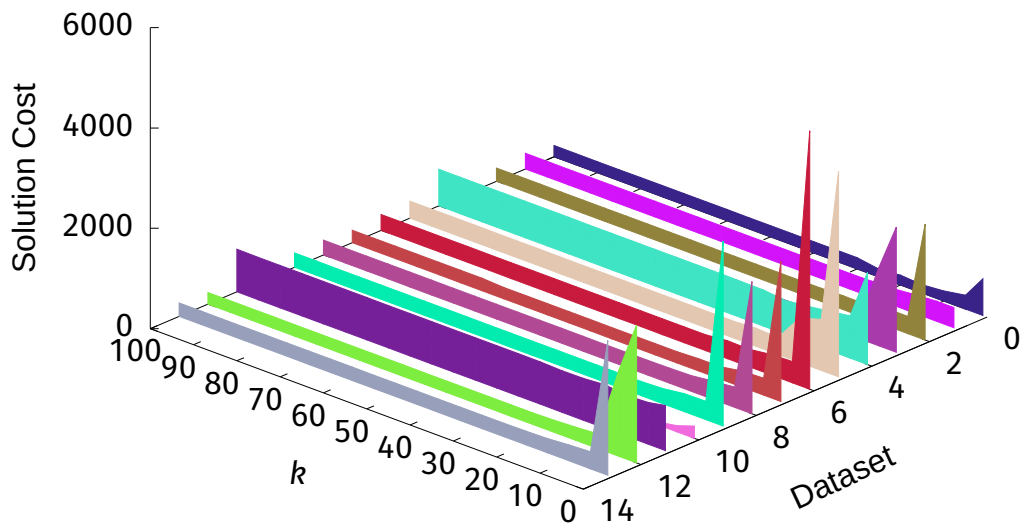


Figure 6.6: The objective/cost function value for the solutions found

7. CONCLUSIONS AND FUTURE DIRECTIONS

Science is the belief in the ignorance of experts.

Richard Feynman

Constraint Programming is quite wide area, and this dissertation contributed to it both in theoretical and practical level. All the implementations were made transparent and available to the open source community.

7.1 Unified random and deterministic heuristics

Our first contribution in this work was to present a well-founded paradigm to exploit both stochastic and deterministic heuristics. Empirical evaluations showed that our hybrid approach can produce better results than fully random or fully deterministic methodologies [71, 72].

In order to achieve this, we approached and used heuristics as a *confidence* measure. By exploiting these heuristic semantics, we were able to produce a new efficient search method, namely PoPS, that can outperform other methodologies. In general, our proposed framework gives the opportunity to exploit “on the fly” whichever heuristic confidence fluctuations occur.

In the future, it will be challenging to parallelize it, as it supports a whole grid of strategies, by concurrently invoking POPSAMPLE with several PieceToCover and conf arguments.

7.2 Distributed Constraint Programming via MapReduce

Another contribution was to consider MapReduce as a framework that is not only well-suited for huge databases but also for the huge search spaces that Constraint Programming explores. We evolved a generic Solver that already supported the definition of custom CSPs and ad hoc search methods. We made it capable of (i) sampling the search trees, (ii) recording the search tree splits into a text file, and (iii) restoring search in a specific search tree part/split. The text file was supplied as input for MapReduce, whose Mappers were Solver instances that could restore search in a given search tree split [70].

In the future, the most promising thing to do is to solve optimization CSPs, which are a fertile ground for even superlinear speedups. In these problems the goal is not to find one or all the solutions, but to get the *best* solution, according

to specific criteria. To achieve this, each mapper-solver can employ the *branch and bound* methodology which dynamically adds a new constraint (each time a solution is found) that the next solution must be better than the one already found. Additionally, when a new MapReduce round begins, all the mappers-solvers will know the best solution found in the previous MapReduce round and adapt their branch and bound strategies accordingly.

Furthermore, we can have superlinear speedups when trying to get just *one* solution of a CSP and not all of them.

7.3 Relaxed constraint propagation: Less is more

An important contribution of this work is to give focus on the weaker consistency levels (bounds consistency – BC) in Constraint Programming and to highlight their advantages over “stronger” consistency levels (arc consistency – AC). If we take it for granted that arc and bounds consistency have both equal asymptotic time complexities, then two questions arise.

1. Why is BC often used in practice in Constraint Programming solvers?
2. When should we prefer BC over AC?

In current bibliography, answers to the first question are scarce and only based on unpublished empirical observations. In any case, one can answer to the first question by conducting experiments and finding examples where BC is more efficient than AC. Indeed, in this work, we experimented with a broad range of official CSPs and found many cases where BC is more efficient in practice.

7.3.1 Predicting the efficiency of relaxed consistency

The second question is more difficult, as it is addressed for every possible ad hoc CSP. Our approach to answer it included the following steps.

- Introduce the algorithms for arc and bounds consistency enforcement and prove that they take the same time in the worst case.
- Introduce a basic backtracking search algorithm and the search tree and search path notions.
- Integrate consistency enforcement algorithms into the backtracking search method.
- Compute the overall time complexity while descending a search tree path and find the differentiations between maintaining AC and BC.
- Project the complexity to traverse a search path to the overall search tree complexity.

Following this approach, we produced some tight upper limits for AC and BC time complexities in the context of search methods. We defined a criterion which, based on the attributes of a CSP, predicts which of the two methodologies is likely to solve it faster.

This new criterion gives us the freedom to select the consistency level (AC or BC) just before solving a specific CSP. We are not obliged to use default consistency levels when we build a Constraint Programming solver anymore. We are now able to tailor the AC vs. BC selection to the particular parameters of each CSP and thus make the overall search process more efficient [73].

In the future, this work can be naturally extended to answer the question why even higher consistency levels than AC are “seldom used in practice” [5]. This is another paradox, as there are a lot of very important publications for sophisticated higher consistency levels. Just like in this work, we should develop criteria about *when* to use higher consistency levels than AC and not completely ignore them.

Another natural future extension of this work will be to compare the maintenance of *generalized* arc and bounds consistencies during search, which are enforced to non-binary constraint networks. In this work, we considered only binary constraints, i.e. only constraints between two variables. This was done for the sake of simplicity, as every constraint involving more than two variables can be converted to binary constraints [81]. After all, the notion of the *arc*, e.g. (X_1, X_2) , includes only two variables.

On the other hand, n -ary constraints with $n > 2$, i.e. constraints that involve more than two variables, are quite common in practice and can be exploited to speed up search. Such constraints are often expressive in the sense that it is more elegant for example to mention $\text{AllDifferent}(X_1, X_2, X_3)$ than $X_1 \neq X_2 \wedge X_2 \neq X_3 \wedge X_3 \neq X_1$.

For n -ary constraints, we enforce either generalized arc consistency (GAC) or generalized bounds consistency. It would be interesting to see if the behavior of maintaining AC vs. BC during search remains the same for their generalized variants.

7.3.2 A new relaxed consistency variant

Finally, we introduced k -bounds-consistency, a parameterized bounds consistency variant [74]. Our new k -bounds-consistency was proved to be more efficient for small k values. Still, the exact specification of the best k is different for each different CSP instance. In the future, it would be interesting to automate the process of finding the “golden mean” k .

Future perspectives also include proposing even looser consistency types for the individual problems with too many variables, for which only local search methods seem nowadays efficient [60]. Except for the domain size, are there any other ways to limit—or augment—the constraint propagation level?

7.3.3 Toward one unified benchmarking

In 1997, Eugene Freuder, a Constraint Programming pioneer, stated that its “holy grail” is that the user simply states the problem and the computer solves it [34]. This, obviously, emphasizes on user experience. Today, after two decades

of theoretical advances, the community still pursues this “holy grail” [35]. If we want to contribute toward this direction in the future, we should integrate and test the existing theory (e.g. about various consistency levels, search methodologies, etc.) into user-friendly solvers and take the decision to use a common testbed with emphasis on real-life CSPs over artificial ones with obfuscated modelings.

In this direction, we can employ the MiniZinc language that allows a single model of a CSP to be solved by multiple different solvers [62]. Furthermore, the CSP instances of the “MiniZinc Challenge” competition can be used as a benchmark e.g. to prove the efficiency of a consistency enforcement algorithm [87].

Nevertheless, in our consistency enforcement experiments we preferred to use CSP instances defined using the XCSP3 format that is more low-level than MiniZinc [14]. There were two reasons behind this choice.

- Many relevant consistency enforcement papers use the instances of the XCSP3 library, and we wanted to be as close to the related work as possible.
- There is an XCSP3-core subset of the XCSP3 language [15]. XCSP3-core contains the most essential constraints, and it is more tailored for competitions and benchmarking than the original XCSP3 and MiniZinc languages. XCSP3-core language and benchmarks are easier to be adopted by a mini-solver used in a research paper.

However, there are some XCSP3 drawbacks that should be addressed too.

- As already mentioned, XCSP3 is more low-level than MiniZinc, which means that it is less user-friendly.
- The default XCSP3 definitions of many CSP instances are obfuscated, as they use table constraints.

To mitigate the above, toward a unified testbed to benchmark the new and old consistency algorithms, a “MiniZinc-core” language and competition should be established.

REFERENCES

- [1] F. Afrati, D. Fotakis, and J. D. Ullman, “Enumerating subgraph instances using Map-Reduce,” in *ICDE 2013*. IEEE, 2013, pp. 62–73.
- [2] A. Ahmeti and N. Musliu, “Min-conflicts heuristic for multi-mode resource-constrained projects scheduling,” in *Genetic and Evolutionary Computation Conference*. ACM, 2018, pp. 237–244.
- [3] M. A. Ayub, K. A. Kalpoma, H. T. Proma, S. M. Kabir, and R. I. H. Chowdhury, “Exhaustive study of essential constraint satisfaction problem techniques based on N-queens problem,” in *ICCIT*. IEEE, 2017, pp. 1–6.
- [4] A. Balafrej, C. Bessiere, and A. Paparrizou, “Multi-armed bandits for adaptive constraint propagation,” in *IJCAI*, 2015, pp. 290–296.
- [5] A. Balafrej, C. Bessiere, A. Paparrizou, and G. Trombettoni, “Adapting consistency in constraint solving,” in *Data Mining and Constraint Programming*, C. Bessiere, L. De Raedt, L. Kotthoff, S. Nijssen, B. O’Sullivan, and D. Pedreschi, Eds. Springer, 2016, pp. 226–253.
- [6] P. Barahona, L. Krippahl, and O. Perriquet, “Bioinformatics: A challenge to constraint programming,” in *Hybrid Optimization*, P. Van Hentenryck and M. Milano, Eds. Springer, 2011, vol. 45, pp. 463–487.
- [7] R. Barták, “Incomplete depth-first search techniques: A short survey,” in *CPDC 2004: Constraint Programming for Decision and Control*. Silesian University of Technology, 2004, pp. 7–14.
- [8] R. Barták and H. Rudová, “Limited assignments: A new cutoff strategy for incomplete depth-first search,” in *SAC 2005: Symposium on Applied Computing*. ACM, 2005, pp. 388–392.
- [9] R. Barták, M. A. Salido, and F. Rossi, “Constraint satisfaction techniques in planning and scheduling,” *Journal of Intelligent Manufacturing*, vol. 21, no. 1, pp. 5–15, 2010.
- [10] C. Bessiere, “Constraint propagation,” in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, pp. 29–83.
- [11] C. Bessiere, “Constraint reasoning,” in *A Guided Tour of Artificial Intelligence Research, Volume II: AI Algorithms*, P. Marquis, O. Papini, and H. Prade, Eds. Springer, 2020, pp. 153–183.

- [12] C. Bessiere, J.-C. Régin, R. H. C. Yap, and Y. Zhang, “An optimal coarse-grained arc consistency algorithm,” *Artificial Intelligence*, vol. 165, no. 2, pp. 165–185, 2005.
- [13] B. Bogaerts, E. Gamba, and T. Guns, “A framework for step-wise explaining how to solve constraint satisfaction problems,” *Artificial Intelligence*, vol. 300, p. 103550, 2021.
- [14] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, “XCSP3: An integrated format for benchmarking combinatorial constrained problems,” arXiv:1611.03398 <https://arxiv.org/abs/1611.03398>, 2016.
- [15] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, “XCSP3-core: A format for representing constraint satisfaction/optimization problems,” arXiv:2009.00514 <https://arxiv.org/abs/2009.00514>, 2020.
- [16] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
- [17] J. L. Bresina, “Heuristic-biased stochastic sampling,” in *AAAI-96*, vol. 1, 1996, pp. 271–278.
- [18] A. A. Bulatov, “A dichotomy theorem for nonuniform CSPs,” in *FOCS 2017*. IEEE, 2017, pp. 319–330.
- [19] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J. P. Warners, “Radio link frequency assignment,” *Constraints*, vol. 4, no. 1, pp. 79–89, 1999.
- [20] M. S. Cherif, D. Habet, and C. Terrioux, “On the refinement of conflict history search through multi-armed bandit,” in *ICTAI*, 2020, pp. 264–271.
- [21] G. Chu, C. Schulte, and P. J. Stuckey, “Confidence-based work stealing in parallel constraint programming,” in *CP 2009*, vol. 5732. Springer, 2009, pp. 226–241.
- [22] V. A. Cicirello and S. F. Smith, “Enhancing stochastic search performance by value-biased randomization of heuristics,” *Journal of Heuristics*, vol. 11, no. 1, pp. 5–34, 2005.
- [23] W. S. Cleveland, E. Grosse, and W. M. Shyu, “Local regression models,” in *Statistical Models in S*, J. M. Chambers and T. J. Hastie, Eds. Chapman & Hall, 1991, ch. 8, pp. 309–376.
- [24] D. A. Cohen and P. G. Jeavons, “The power of propagation: When GAC is enough,” *Constraints*, vol. 22, no. 1, pp. 3–23, 2017.
- [25] W. Cohen, R. Greiner, and D. Schuurmans, “Probabilistic hill-climbing,” in *Computational Learning Theory and Natural Learning Systems*, vol. 2. Cambridge: MIT Press, 1994, pp. 171–181.
- [26] C. M. Dasari and R. Bhukya, “MapReduce paradigm: DNA sequence clustering based on repeats as features,” *Expert Systems*, p. e12827, 2021.

- [27] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *OSDI*. USENIX, 2004, pp. 137–149.
- [28] D. Delahaye, S. Chaimatanan, and M. Mongeau, “Simulated annealing: From basics to applications,” in *Handbook of Metaheuristics*, M. Gendreau and J.-Y. Potvin, Eds. Springer, 2019, pp. 1–35.
- [29] S. Dev Gupta, B. Genc, and B. O’Sullivan, “Explanation in constraint satisfaction: A survey,” in *IJCAI*, 2021, pp. 4400–4407.
- [30] F. Es-Sabery, K. Es-Sabery, J. Qadir, B. Sainz-De-Abajo, A. Hair, B. García-Zapirain, and I. De La Torre-Díez, “A MapReduce opinion mining for COVID-19-related tweets classification using enhanced ID3 decision tree classifier,” *IEEE Access*, vol. 9, pp. 58 706–58 739, 2021.
- [31] J.-G. Fages and C. Prud’Homme, “Making the first solution good!” in *ICTAI*, 2017, pp. 1073–1077.
- [32] “First XCSP3 competition,” <http://xcsp.org/call2017.pdf>, 2017.
- [33] M. Fischetti, M. Monaci, and D. Salvagnin, “Self-splitting of workload in parallel computation,” in *CPAIOR 2014*, vol. 8451. Springer, 2014, pp. 394–404.
- [34] E. C. Freuder, “In pursuit of the holy grail,” *Constraints*, vol. 2, no. 1, pp. 57–61, 1997.
- [35] E. C. Freuder, “Progress towards the holy grail,” *Constraints*, vol. 23, no. 2, pp. 158–171, 2018.
- [36] “Gecode: Generic constraint development environment,” <http://www.gecode.org>, 2019.
- [37] I. P. Gent, I. Miguel, P. Nightingale, C. McCreesh, P. Prosser, N. C. A. Moore, and C. Unsworth, “A review of literature on parallel constraint solving,” *Theory and Practice of Logic Programming*, vol. 18, no. 5-6, pp. 725–758, 2018.
- [38] I. P. Gent and T. Walsh, “CSPLIB: A benchmark library for constraints,” in *CP 1999*, vol. 1713. Springer, 1999, pp. 480–481, <http://CSPLib.org>.
- [39] I. P. Gent and T. Walsh, “CSPLIB: Twenty years on,” arXiv:1909.13430 <https://arxiv.org/abs/1909.13430>, 2019.
- [40] M. Gergatsoulis, C. Nomikos, E. Kalogeros, and M. Damigos, “An algorithm for querying linked data using Map-Reduce,” in *6th International Conference, Globe 2013: Data Management in Cloud, Grid and P2P Systems*, ser. LNCS, vol. 8059. Springer, 2013, pp. 51–62.
- [41] J. Ginsberg, M. H. Mohebbi, R. S. Patel, L. Brammer, M. S. Smolinski, and L. Brilliant, “Detecting influenza epidemics using search engine query data,” *Nature*, vol. 457, no. 7232, pp. 1012–1014, 2009.

- [42] M. L. Ginsberg and W. D. Harvey, "Iterative broadening," *Artificial Intelligence*, vol. 55, no. 2-3, pp. 367–383, 1992.
- [43] C. P. Gomes, B. Selman, N. Crato, and H. Kautz, "Heavy-tailed phenomena in satisfiability and constraint satisfaction problems," *Journal of Automated Reasoning*, vol. 24, no. 1-2, pp. 67–100, 2000.
- [44] A. Grasas, A. A. Juan, J. Faulin, J. de Armas, and H. Ramalhinho, "Biased randomization of heuristics using skewed probability distributions: A survey and some applications," *Computers & Industrial Engineering*, vol. 110, pp. 216–228, 2017.
- [45] D. Habet and C. Terrioux, "Conflict history based heuristic for constraint satisfaction problem solving," *Journal of Heuristics*, vol. 27, no. 6, pp. 951–990, 2021.
- [46] H. H. Hoos and E. Tsang, "Local search methods," in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, ch. 5, pp. 135–167.
- [47] I. Howell, R. J. Woodward, B. Y. Choueiry, and C. Bessiere, "Solving Sudoku with consistency: A visual and interactive approach," in *IJCAI*, 2018, pp. 5829–5831.
- [48] B. Jafari and M. Mouhoub, "Heuristic techniques for variable and value ordering in CSPs," in *Genetic and Evolutionary Computation Conference*. ACM, 2011, pp. 457–464.
- [49] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.
- [50] A. Korikov and J. C. Beck, "Counterfactual explanations via inverse constraint programming," in *CP 2021*, vol. 210, 2021, pp. 35:1–35:16.
- [51] V. Koukis, C. Venetsanopoulos, and N. Koziris, "~okeanos: Building a cloud, cluster by cluster," *IEEE Internet Computing*, vol. 17, no. 3, pp. 67–71, 2013.
- [52] V. Koukis, C. Venetsanopoulos, and N. Koziris, "Synnefo: A complete cloud stack over Ganeti," *USENIX ;login.*, vol. 38, no. 5, pp. 6–10, 2013.
- [53] A. Lallouet, Y. Moinard, P. Nicolas, and I. Stéphan, "Logic programming," in *A Guided Tour of Artificial Intelligence Research, Volume II: AI Algorithms*, P. Marquis, O. Papini, and H. Prade, Eds. Springer, 2020, pp. 83–113.
- [54] H. Li, R. Li, and M. Yin, "Saving constraint checks in maintaining coarse-grained generalized arc consistency," *Neural Computing and Applications*, vol. 31, no. 1, pp. 499–508, 2019.
- [55] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, no. 1, pp. 99–118, 1977.
- [56] A. K. Mackworth, "Constraint-based design of embedded intelligent systems," *Constraints*, vol. 2, no. 1, pp. 83–86, 1997.

- [57] P. Marquis, O. Papini, and H. Prade, Eds., *A Guided Tour of Artificial Intelligence Research, Volume II: AI Algorithms*. Springer, 2020.
- [58] B. McCollum, A. Schaerf, B. Paechter, P. McMullan, R. Lewis, A. J. Parkes, L. Di Gaspero, R. Qu, and E. K. Burke, “Setting the research agenda in automated timetabling: The second international timetabling competition,” *INFORMS Journal on Computing*, vol. 22, no. 1, pp. 120–130, 2010.
- [59] S. Melenli and A. Topkaya, “Real-time maintaining of social distance in COVID-19 environment using image processing and big data,” in *Innovations in Intelligent Systems and Applications*. IEEE, 2020, pp. 1–5.
- [60] L. Michel and P. Van Hentenryck, “Constraint-based local search,” in *Handbook of Heuristics*, R. Martí, P. Panos, and M. G. C. Resende, Eds. Springer, 2017, pp. 1–38.
- [61] U. Montanari, “Networks of constraints: Fundamental properties and applications to picture processing,” *Information Sciences*, vol. 7, pp. 95–132, 1974.
- [62] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, “MiniZinc: Towards a standard CP modelling language,” in *CP 2007*. Springer, 2007, pp. 529–543.
- [63] V. Pandey and P. Saini, “Constraint programming versus heuristic approach to MapReduce scheduling problem in Hadoop YARN for energy minimization,” *The Journal of Supercomputing*, vol. 77, no. 7, pp. 6788–6816, 2021.
- [64] R. Patan, S. Kallam, A. H. Gandomi, T. Hanne, and M. Ramachandran, “Gaussian relevance vector MapReduce-based annealed Glowworm optimization for big medical data scheduling,” *Journal of the Operational Research Society*, pp. 1–12, 2021.
- [65] J. Petke, *Bridging Constraint Satisfaction and Boolean Satisfiability*. Springer, 2015.
- [66] A. Popescu, S. Polat-Erdeniz, A. Felfernig, M. Uta, M. Atas, V.-M. Le, K. Pilsl, M. Enzelsberger, and T. N. T. Tran, “An overview of machine learning techniques in constraint solving,” *Journal of Intelligent Information Systems*, vol. 58, no. 1, pp. 91–118, 2022.
- [67] N. Pothitos, “NAXOS SOLVER,” <https://github.com/pothitos/naxos>, 2021.
- [68] N. Pothitos, G. Kastrinis, and P. Stamatopoulos, “Constraint propagation as the core of local search,” in *SETN 2012*, ser. LNCS (LNAI), vol. 7297. Springer, 2012, pp. 9–16.
- [69] N. Pothitos and P. Stamatopoulos, “Flexible management of large-scale integer domains in CSPs,” in *SETN 2010*, ser. LNCS (LNAI), vol. 6040. Springer, 2010, pp. 405–410.

- [70] N. Pothitos and P. Stamatopoulos, “Constraint Programming MapReduce’d,” in *SETN 2016*. ACM, 2016, pp. 5:1–5:4.
- [71] N. Pothitos and P. Stamatopoulos, “Piece of Pie Search: Confidently exploiting heuristics,” in *SETN 2016*. ACM, 2016, pp. 8:1–8:8.
- [72] N. Pothitos and P. Stamatopoulos, “Building search methods with self-confidence in a constraint programming library,” *International Journal on Artificial Intelligence Tools*, vol. 27, no. 4, pp. 1 860 003:1–1 860 003:34, 2018.
- [73] N. Pothitos and P. Stamatopoulos, “The dilemma between arc and bounds consistency,” *International Journal of Intelligent Systems*, vol. 35, no. 10, pp. 1467–1491, 2020.
- [74] N. Pothitos, P. Stamatopoulos, and K. Zervoudakis, “Course scheduling in an adjustable constraint propagation schema,” in *ICTAI 2012*, vol. 1. IEEE, 2012, pp. 335–343.
- [75] P. Prosser and C. Unsworth, “Limited discrepancy search revisited,” *Journal of Experimental Algorithmics*, vol. 16, pp. 1.6:1–1.6:18, 2011.
- [76] J.-F. Puget and P. Van Hentenryck, “A constraint satisfaction approach to a circuit design problem,” *Journal of Global Optimization*, vol. 13, no. 1, pp. 75–93, 1998.
- [77] J.-C. Régin, M. Rezgoui, and A. Malapert, “Embarrassingly parallel search,” in *CP 2013*, vol. 8124. Springer, 2013, pp. 596–610.
- [78] J.-C. Régin, M. Rezgoui, and A. Malapert, “Une adaptation simple et efficace du modèle MapReduce pour la programmation par contraintes,” in *JFPC 2013: 9^{es} Journées Francophones de Programmation par Contraintes, Aix-en-Provence*, 2013, pp. 289–298.
- [79] J.-C. Régin, M. Rezgoui, and A. Malapert, “Improvement of the embarrassingly parallel search for data centers,” in *CP 2014*, vol. 8656. Springer, 2014, pp. 622–635.
- [80] C. C. Rolf and K. Kuchcinski, “Combining parallel search and parallel consistency in constraint programming,” in *TRICS: Techniques for Implementing Constraint programming Systems*, 2010, pp. 38–52.
- [81] F. Rossi, C. Petrie, and V. Dhar, “On the equivalence of constraint satisfaction problems,” in *ECAI*, 1990, pp. 550–556.
- [82] F. Rossi, P. van Beek, and T. Walsh, Eds., *Handbook of Constraint Programming*. Elsevier, 2006.
- [83] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2021, ch. 6, pp. 180–207.

- [84] D. Sharma, V. Singh, and C. Sharma, “GA-based scheduling of FMS using roulette wheel selection process,” in *International Conference on Soft Computing for Problem Solving*, vol. 131. Springer, 2012, pp. 931–940.
- [85] B. M. Smith, “Modelling,” in *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Elsevier, 2006, pp. 377–406.
- [86] K. Stergiou, “Adaptive constraint propagation in constraint satisfaction: Review and evaluation,” *Artificial Intelligence Review*, vol. 54, no. 7, pp. 5055–5093, 2021.
- [87] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, and J. Fischer, “The MiniZinc challenge 2008–2013,” *AI Magazine*, vol. 35, no. 2, pp. 55–60, 2014.
- [88] E. Tsang, *Foundations of Constraint Satisfaction*. Norderstedt: Books on Demand, 2014.
- [89] M. Veksler and O. Strichman, “Learning general constraints in CSP,” *Artificial Intelligence*, vol. 238, pp. 135–153, 2016.
- [90] R. Walker, “The guts of a new machine,” *NYTimes.com*, Nov. 30 2014, which quotes S. Jobs.
- [91] T. Walsh, “Depth-bounded discrepancy search,” in *IJCAI*, vol. 2. San Francisco: Morgan Kaufmann, 1997, pp. 1388–1393.
- [92] D. Waltz, “Understanding line drawings of scenes with shadows,” in *The Psychology of Computer Vision*. McGraw-Hill, 1975, pp. 19–91.
- [93] R. Wang and R. H. C. Yap, “Arc consistency revisited,” in *CPAIOR 2019*. Springer, 2019, pp. 599–615.
- [94] H. Watez, F. Koriche, C. Lecoutre, A. Paparrizou, and S. Tabary, “Learning variable ordering heuristics with multi-armed bandits and restarts,” in *ECAI 2020*, vol. 325, 2020, pp. 371–378.
- [95] H. Watez, C. Lecoutre, A. Paparrizou, and S. Tabary, “Refining constraint weighting,” in *ICTAI*, 2019, pp. 71–77.
- [96] I. Wegener, *Complexity Theory*. Springer, 2005, ch. 2, p. 20.
- [97] R. Wilcox, “The regression smoother LOWESS: A confidence band that allows heteroscedasticity and has some specified simultaneous probability coverage,” *Journal of Modern Applied Statistical Methods*, vol. 16, no. 2, pp. 29–38, 2017.
- [98] R. J. Woodward, B. Y. Choueiry, and C. Bessiere, “A reactive strategy for high-level consistency during search,” in *IJCAI*, 2018, pp. 1390–1397.
- [99] W. Xia and R. H. C. Yap, “Learning robust search strategies using a bandit-based approach,” in *AAAI-18*, 2018, pp. 6657–6665.

- [100] Y. Xu, D. Stern, and H. Samulowitz, “Learning adaptation to solve constraint problems,” in *LION 3: 3rd International Conference on Learning and Intelligent Optimization*. Springer, 2009.
- [101] R. H. C. Yap, W. Xia, and R. Wang, “Generalized arc consistency algorithms for table constraints: A summary of algorithmic ideas,” in *AAAI-20*, vol. 34, no. 09, 2020, pp. 13 590–13 597.
- [102] D. Zhuk, “A proof of CSP dichotomy conjecture,” in *FOCS 2017*. IEEE, 2017, pp. 331–342.