Università degli Studi di Ferrara

DOTTORATO DI RICERCA IN SCIENZE DELL'INGEGNERIA

CICLO XXXIV

COORDINATORE Prof. Trillo Stefano

Improving Reasoning in Constraint Logic Programming: an Application to Route Planning and Qualitative Temporal Reasoning Problems

Settore Scientifico Disciplinare ING-INF/05

Dottorando	**Tutore**
Dott. Bertagnon Alessandro	Prof. Gavanelli Marco

Anni 2018/2021

Acknowledgements

I would like to express my sincere gratitude to my supervisor Prof. Marco Gavanelli for the continuous guidance, support, patience and encouragement. I am really lucky to have him as a supervisor and I will be forever grateful.

I am also grateful to Prof. Guido Sciavicco for his collaboration during my PhD.

Many thanks also to my colleagues and co-authors Elena Bellodi, Alessandro Passantino, Stefano Trevisani and Riccardo Zese for their contributions to the research presented in this thesis.

I would also like to thank my office mates Damiano, Francesco, Arnaud and Michele for being part of this experience and for their support for both research and non-research activities.

I would like to warmly thank the referees for their willingness to read and comment my thesis. Their insightful remarks and comments help me improve my work.

Finally, a huge thanks to my family, my mother Marisa and my father Francesco for always being there, supporting me through ups and downs.
Thanks also to Claudia, to all my friends, and to all the people I met during my PhD who made these three years unforgettable.

Contents

1 Introduction

Contents

The concept of *constraint* is ubiquitous, and we have concrete examples of it in everyday life. We can think of a constraint as a condition that limits our ability to make decisions.

Consider, for example, the finite number of hours in a day, which places a limit on the number of activities we may decide to do, or the power limit of our homes which prevents us from turning on all our electrical appliances at the same time. The list of examples could go on but, we assume that at this point the reader of this thesis might have come up with other examples based on personal experiences.

A decision problem that bases its formulation on constraints is called Constraint Satisfaction Problem (CSP). In this scenario, constraints represent some functional relationships between the problem variables and other parameters satisfying certain physical phenomenon and certain resource limitations. Solving a CSP consists of making a series of decisions such that all constraints are satisfied.

Basically, every day we face and solve CSPs without even realising it and without the need to use sophisticated algorithms or particularly powerful computers. However, the situation becomes more complicated when the problems to solve become bigger, i.e., the number of decisions to be taken increases.

Many problems in the field of Artificial Intelligence (AI) can be classified as CSPs; and it was in the AI area that the first solving techniques for CSPs were developed. In particular, the first solvers employed logic programming for problem modelling, i.e., a declarative approach. Formulating problems in a declarative way in terms of constraints is natural to the user because it only requires stating *what* must be satisfied without having to say *how* it must be satisfied. This way of addressing CSPs led to the birth of the programming paradigm known as Constraint Logic Programming (CLP) [108]. Later, the declarative approach for constraint modelling was also extended to imperative programming languages, and the corresponding paradigm was called Constraint Programming (CP) [164].

Furthermore, for many problems it is not sufficient to find a solution that satisfies the constraints, but among all the feasible solutions it is necessary to find the best one. The definition of the best solution may depend on several factors, but in general it is that solution which minimises the quantities representing costs and/or maximises those representing profits. These problems, in order to be distinguished from basic CSPs, are identified as Constraint Optimization Problems (COPs).

Constraint solving techniques, also known as constraint reasoning algorithms, can be divided into two distinct and orthogonal strategies: inference (also known as constraint propagation) and search. Both of these two strategies, which will be discussed in more detail in Chapter 2, are important in the resolution of CSPs and COPs, and in fact, are very often combined to achieve the best performance overall.

The computational complexity of CSPs and COPs, in their general form, is NP-HARD. This makes it very important to have algorithms that are effective in practice, for example by taking advantage of tractable cases of such problems, by exploiting constraints to reduce the search space, or by adopting intelligent heuristics.

Consequently, the development of new effective inference and/or search algorithms is of great importance to allow progress in the field.

1.1 Motivation

Constraint Satisfaction Problems and Constraint Optimization Problems are ubiquitous in many applications in engineering and industry. A list (by far not exhaustive) of application domains of CP (and thus of CLP) includes: resource allocation, product configuration, network problems, bioinformatics, automated planning and scheduling, temporal reasoning and vehicle routing.

Despite of this categorisation, it is therefore normal for a lot of real-world applications to belong to more than one application domain due to their characteristics.

In the research presented in this thesis we will focus on temporal reasoning and vehicle route planning as they arise in many applications used every day.

Vehicle route planning is a process that involves planning the route from one location to another and generally involves creating routes for vehicles to reach a group of customers at a minimal cost. The problems in this area are very often related to supply chain optimisation. The movement of goods and people plays a central role in the costs sustained by companies around the world and even small improvements can lead to enormous cost savings and thus higher profits. Of all the problems falling into this category, the one formulated first is the Traveling Salesperson Problem (TSP).

Traveling Salesperson Problem

Given a weighted graph the TSP requires to compute the minimum cost cycle that visits each vertex exactly once. The name *Traveling Salesperson Problem* comes from the problem's most famous formulation: *"A salesman has to visit a set of cities, each of which must be visited only once, and he wants to minimize the length of the tour".* The TSP is one of the best-known problems in computer science and its applications go beyond vehicle route planning. Some of the general applications of the TSP include: genome sequencing [52, 97, 26, 2], drilling problems [98, 142], aiming telescopes [182], x-ray crystallography [38], data clustering [144, 130], pattern-cutting [141], order-picking problem in warehouses [169], and many others.

Due to its wide notoriety some special cases of the TSP have been studied in the literature over the years; special cases relate to the fact that additional assumptions are imposed on the available data. A well-known special case is the Euclidean Traveling Salesperson Problem (ETSP). In the ETSP each node of the weighted graph is identified by its coordinate on the plane and the Euclidean distance is used as cost function. Although more information is available in the ETSP than in the general TSP, it maintains the same complexity (NP-HARD) as the more general case [81].

The ETSP maintains an important number of applications and indeed many benchmarks for the TSP, see for example the TSPLIB [172], relate precisely to it.

Despite its widespread diffusion, no specialized constraint propagation algorithms for solving ETSP in CP has ever been proposed so far. The usual way to tackle Euclidean TSPs is to compute the distance matrix and address the problem as a general TSP, without using any of the additional information available in this case (e.g., node coordinates and geometrical concepts like: straight line segments, angles, etc.).

Qualitative temporal reasoning

The concept of time is ubiquitous, and the explicit representation and reasoning of time is a major problem in many areas of AI. Human perception and understanding of the real world deeply embody the concept of time. Everything appears related due to its temporal relation. Events occur temporally in relation to one another. When the constraints of a problem concern the temporal relations between events, the problem is identified as temporal CSPs. Temporal CSPs have been studied extensively because of their importance in applications such as: natural language processing [187], scheduling [156], planning [5], database theory [123], medical diagnosis [155], circuit design [193], archaeology [95, 118], and genetics [28]. There are two approaches for temporal reasoning: a *quantitative* one and a *qualitative* one. Qualitative temporal relations allow for representing time associated with events or facts in a somewhat indefinite manner. As an example, consider the sentence "Alice had breakfast and then met Bob", there is no reference to what time the two actions occurred ("had breakfast", "met Bob") or how long they lasted, the only information provided is the order in which the actions took place. Quantitative temporal relations allow for representing more information about events, for example in the sentence "Alice took the bus at 9.30 a.m., then met Bob at 9.45 a.m." we know precisely that the time it took Alice to reach Bob was 15 minutes.

Qualitative temporal reasoning is an area that has greatly benefited from CP techniques since James F. Allen proposed an algebra, in 1983, that later became known as Allen's Interval Algebra ($\mathcal{IA}$) [6]. $\mathcal{IA}$ is a qualitative, interval-based algebra, where time is interpreted as a continuous line. Later, an extension of the $\mathcal{IA}$ was proposed and called Branching Interval Algebra ($\mathcal{BA}$) [168]. The $\mathcal{BA}$, as the name suggests, is a tree-like extension of the linear formalism because time is no longer seen as a straight line but rather as a tree, which can branch in the future, but its branches, once created, can never merge. So, $\mathcal{BA}$ introduces the concept of *alternative* or *choice* to reason about alternative futures. $\mathcal{BA}$ also has many potential applications in different areas of AI; for example, in planning with alternatives [143, 152, 197], in which different plans have to be taken into account in the design phase, or in the automatic generation of narratives [176] and in the formal verification of parallel programs [179].

Unfortunately, the consistency problem of $\mathcal{BA}$ is NP-HARD, as in the linear case. In the linear case, we know every tractable fragment of the full algebra [124], while in the branching case the entire landscape of tractable fragments is still unknown. Unlike the linear case, however, there has not been much effort to discover potentially tractable fragments. Therefore, identifying tractable fragments is particularly important to efficiently solve the problems expressed in $\mathcal{BA}$. In fact, tractable fragments of an algebra can be used as heuristics in search algorithms to improve their efficiency.

1.2 Contribution

The main contributions of this thesis are:

- We present the first attempt, to the best of our knowledge, to exploit the geometric information in order to achieve a stronger constraint propagation in CP when solving ETSPs.

- We show that the inference performed by the algorithms we developed for the ETSP is not subsumed by that of state-of-the-art algorithms in CP for solving the TSP.
- We study the computational complexity of some of the newly introduced constraint propagation algorithms and show how machine learning techniques can be combined with CP techniques to increase overall solving performance.
- We developed algorithms to study $\mathcal{BA}$ fragments in order to assess their tractability (or prove that they are intractable). In particular, our research focused on four interesting fragments named: $\mathcal{BA}_{convex}$, $\mathcal{BA}_{point}$, $\mathcal{BA}_{lin}$, $\mathcal{BA}_{Horn}$.
- We propose an enhanced version of a search algorithm for the full $\mathcal{BA}$ that takes advantage from tractable fragments to improve its computational performance.
- We experimentally evaluated and showed that the proposed search algorithm can be used to effectively solve problems expressed in $\mathcal{BA}$.

1.3 Publications

This thesis is based on several peer-reviewed publications:

- Alessandro Bertagnon and Marco Gavanelli. "Improved Filtering for the Euclidean Traveling Salesperson Problem in CLP(FD)". in: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 2020, pp. 1412–1419 Scopus: 2-s2.0-85092671488
- Elena Bellodi, Alessandro Bertagnon, Marco Gavanelli, and Riccardo Zese. "Improving the Efficiency of Euclidean TSP Solving in Constraint Programming by Predicting Effective Nocrossing Constraints". In: *Joint Proceedings of the 8th Italian Workshop on Planning and Scheduling and the 27th International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion co-located with AIxIA 2020, Online Event, November 25-27, 2020.* Ed. by Riccardo De Benedictis et al. Vol. 2745. CEUR Workshop Proceedings. CEUR-WS.org, 2020 Scopus: 2-s2.0-85096984335
- Elena Bellodi, Alessandro Bertagnon, Marco Gavanelli, and Riccardo Zese. "Improving the Efficiency of Euclidean TSP Solving in Constraint Programming by Predicting Effective Nocrossing Constraints". In: *AIxIA 2020 - Advances in Artificial Intelligence - XIXth International Conference of the*

Italian Association for Artificial Intelligence, Virtual Event, November 25-27, 2020, Revised Selected Papers. Ed. by Matteo Baldoni and Stefania Bandini. Vol. 12414. Lecture Notes in Computer Science. Springer, 2020, pp. 318–334 Scopus: 2-s2.0-85111397068

- Alessandro Bertagnon. “Constraint Programming Algorithms for Route Planning Exploiting Geometrical Information”. In: *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020.* Ed. by Francesco Ricca et al. Vol. 325. EPTCS. 2020, pp. 286–295 Scopus: 2-s2.0-85092617425

- Alessandro Bertagnon, Marco Gavanelli, Alessandro Passantino, Guido Sciavicco, and Stefano Trevisani. “The Horn Fragment of Branching Algebra”. In: *Proc. of the 27th International Symposium on Temporal Representation and Reasoning.* Vol. 178. LIPIcs. 2020, 5:1–5:16 Scopus: 2-s2.0-85091655245

- Alessandro Bertagnon, Marco Gavanelli, Guido Sciavicco, and Stefano Trevisani. “On (Maximal, Tractable) Fragments of the Branching Algebra”. In: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020.* Ed. by Francesco Calimeri, Simona Perri, and Ester Zumpano. Vol. 2710. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 113–126 Scopus: 2-s2.0-85095837937

- Alessandro Bertagnon, Marco Gavanelli, Alessandro Passantino, Guido Sciavicco, and Stefano Trevisani. “Branching interval algebra: An almost complete picture”. In: *Information and Computation* 281 (2021), p. 104809. ISSN: 0890-5401 Scopus: 2-s2.0-85118261560

1.4 Outline

The content of this thesis is organized as follows. Chapter 2 provides background information on Constraint (Logic) Programming. Chapter 3 presents the Euclidean Traveling Salesperson Problem, introduce inference algorithms based on geometrical information and then propose a way to improve the overall solving performance using machine learning techniques. Chapter 4 presents interval and branching algebras for qualitative temporal reasoning, introduce new expressive but tractable fragments of the branching algebra and shows how to use them to improve the performance of a search algorithm. Finally, we conclude our work in Chapter 5.

2 Constraint (Logic) Programming

Contents

Constraint Programming (CP) is a programming paradigm used to model and solve CSPs, which is a well-known class of problems in the field of Artificial Intelligence (AI) and Operations Research. Simplifying, a CSP is a representation of a problem which consist of: a set of variables, each variable has associated a set of possible values that it can assume, and a number of relations, called constraints, that restrict the set of values that different variables can assume simultaneously. A CSP is considered solved when each variable has associated a value, from its set of possible values, that satisfies all the constraints. The concept of *constraint* was first introduced in Logic Programming (LP), leading to the birth of Constraint Logic Programming (CLP) [109]. Although the declarative approach for constraint modelling was later extended to imperative programming languages, CLP still maintains considerable importance in the resolution of CSPs. This chapter is intended to provide a brief introduction to terminology and symbols of CP and

CLP that will be frequently mentioned in this thesis. For more details, many excellent books are available, such as [104, 190, 58, 15, 80, 178].

2.1 Constraint Satisfaction (Optimization) Problems

2.1.1 Definitions

Definition 2.1.1. (Constraint Satisfaction Problem) A Constraint Satisfaction Problem (CSP) is a triple $\mathscr{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X}$ is a set of n decision variables $\{x_1, x_2, \dots, x_n\}$, $\mathcal{D}$ is a set of domains $\{D_1, D_2, \dots, D_n\}$ and $\mathcal{C}$ a set of constraints $\{c_1, c_2, \dots, c_m\}$. Each domain D_i consist in a set of all possible values that can be assigned to the variable x_i. Each constraint c_i consists of a pair $\langle R_i, S_i \rangle$ where R_i is a relation between the variables S_i participating in the constraint. The set S_i is called the *scope* of R_i. R_i results in a subset of the cartesian product of the domain of the variables in S_i.

A decision variable $x_i \in \mathcal{X}$ is *fixed*, or *instantiated* or *assigned*, by its domain D_i, if $|D_i| = 1$. To represent different sets of domains (implicitly) referred to the same set of decision variables, we will use the notation $\mathcal{D}_1, \dots, \mathcal{D}_j$. This will be useful to represent how the set of domains $\mathcal{D}$ in a CSP varies as a result of the change of one or more variable domains $D_i \in \mathcal{D}$. To represent the domain of the decision variable x_i, indicating which set of domains $\mathcal{D}_j$ we are referring to, we write $\mathcal{D}_j(x_i)$. A set of domains $\mathcal{D}_i$ is said to be *stronger* than a set of domains $\mathcal{D}_j$, written $\mathcal{D}_i \sqsubseteq \mathcal{D}_j$, if $\mathcal{D}_i(x_k) \subseteq \mathcal{D}_j(x_k)$ for all $x_k \in \mathcal{X}$.

A problem state is defined by an *instantiation* of a subset of its variables. An instantiation of a set of variables $\{x_1, \dots, x_k\}$ is defined as a k-tuple $(0 < k \leq n)$ of ordered pairs $\mathcal{I} = (\langle x_1, a_1 \rangle, \dots, \langle x_k, a_k \rangle)$ where each pair $\langle x, a \rangle$ is an assignment of the value a to the variable x where $a \in D(x)$. We also use the notation $\mathcal{I} = (x_1 = a_1, \dots, x_k = a_k)$ or $\mathcal{I} = (a_1, \dots, a_k)$. An instantiation, for a CSP $\mathscr{P}$, is called *complete* if it assigns a value to each variable in $\mathcal{X}$ and *consistent* (or *feasible*) if it satisfies all the constraints $\mathcal{C}$ of $\mathscr{P}$. An instantiation which is complete and consistent is also called a *solution* of $\mathscr{P}$. CSPs for which at least one solution exists are called *consistent* (or satisfiable), while instances that do not have any solutions are called *inconsistent* (or unsatisfiable).

Different types of CSPs are distinguished according to the type of domains assumed by their variables. In the following, we will focus only on *finite discrete domains*, which are the ones of interest in this thesis. A CSP $\mathscr{P}$ is a finite discrete CSP if all decision variables in $\mathscr{P}$ have discrete and finite domains.

If a constraint involves only one decision variable, the size of its scope is 1, the constraint is called a *unary* constraint (e.g., $x_i \neq 3$). Whereas if a constraint involves two variables, the size of its scope is 2, it is called a *binary* constraint (e.g., $x_i + x_j > 0$). If all the constraints of a CSP are unary or binary, the CSP is called *binary CSP*. A binary CSP can be represented by a constraint graph: the vertices of the graph correspond to the decision variables and each constraint is represented by an edge connecting the involved variables. If the arity of the constraints is not limited, so p-ary constraints are present in the CSP, then a hypergraph is required to represent the CSP with a hyperarc for each p-ary constraint connecting the p vertices involved. A constraint that involves an arbitrary subset of variables is called a *global constraint*.

Unless otherwise specified, for a CSP it is assumed that all solutions are equally good. For some applications, however, some solutions are better than others. The task in such problems is to find optimal solution(s), where optimality is defined in terms of some application-specific functions. To identify which solution is "better" than the others, a function f (called *cost function* or *optimization function* or *objective function*) is used which associates a numerical value with each solution. The value assumed by the function should be minimised if it is a cost or maximised if it is a profit. To be distinguished from CSPs, these problems are called Constraint Optimization Problems.

Definition 2.1.2. (Constraint Optimization Problem) A Constraint Optimization Problem (COP) is a pair $\langle \mathscr{P}, f \rangle$ where $\mathscr{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a CSP and f a function $f : D(x_1) \times \cdots \times D(x_n) \to \mathbb{R}$ which associates a value to every solution of the problem. A solution s of $\mathscr{P}$ is also an *optimal solution* of the minimization (resp. maximization) problem if and only if there is no solution s' such that $f(s') < f(s)$ (resp. $f(s) < f(s')$).

If all domains D_i are finite, then the finite *search space* containing all possible instantiation for a CSP or COP is $\Omega = \bowtie_i D_i$ where $\bowtie$ is the join operator of relational algebra. The search space Ω could, at least theoretically, be enumerated and every n-tuple could be tested to check if it actually represents a solution, but this strategy, however, turns out to be too poorly performing due to the large size assumed by Ω even for problems with a small number of decision variables (and/or small domains).

To improve solving performance two distinct and orthogonal strategies can be used: *search* and *inference* (also known as *constraint propagation*). Search (systematically) explores Ω in order to find the solution of the CSP, eliminating infeasible subspaces whenever they are detected. Inference techniques instead take advantage of constraint propagation to eliminate large subspaces of Ω that do not contain feasible instantiations according to the constraints. When solving a CSP, the classic way to proceed is to alternate between these two strategies.

Algorithms for solving CSPs are also divided according to their ability to find solutions into: *complete* algorithms and *incomplete* algorithms. An algorithm is said to be complete if it guarantees to find a solution if one exists; it can also be used to prove that a CSP has no solution or to find the optimal solution of a COP. Incomplete algorithms, on the other hand, cannot guarantee that they find the optimal solution and if they do not find a solution it is not necessarily the case that the problem has no solution.

2.1.2 Search

The backtracking search algorithm is the fundamental complete search method for exploring the search space. The simplest backtrack search algorithm constructs the solution of the problem by incrementally choosing values for variables until it reaches a *dead-end*, i.e., a non-complete instantiation that cannot be further extended in a consistent manner while still satisfying the constraints of the problem.

The backtracking search is often based on a representation of the search space Ω as a tree, which is called *search tree*. Backtracking search is nothing more than a depth-first exploration of the search tree that is generated as the search proceeds.

In the search tree the root node (level 0) is the empty set of assignments while a node at level j is a set of assignments $\{x_1 = a_1, \ldots, x_j = a_j\}$. At each node of the search tree a variable not yet assigned is selected and the branches exiting that node represent all possible ways of extending the current instantiation.

The method used to extend a node of the search tree is called a *branching strategy* and several alternatives have been proposed and examined in the literature.

If, after choosing a branch for the variable, a constraint check fails, another branch is selected following the branching strategy. If there are no more branches available for the current variable, the assignment cannot be extended and, a dead-end occurs.

When a dead-end is reached, the algorithm undoes the last choice made by discarding the current assignment (*backtrack*) and attempts a different choice. The whole subtree rooted where the dead-end occurred is skipped (*pruned*). Usually, the procedure is iterated in a systematic way ensuring that every possible solution is considered at some point.

Backtracking search is more efficient than the trivial enumeration of all possible instantiation of the search space Ω because at each step it is checked whether the constraints are satisfied instead of waiting for the generation of a complete instantiation (candidate solution).

Basic backtracking search backtracks chronologically to the last choice, and this can lead to *thrashing* [39]. Thrashing is the repeated exploration of subtrees that cause the overall search tree to fail, differing only by assignments to variables not responsible for the failure. Since there are an exponential number of such subtrees, thrashing has a non-negligible cost in the running time of the backtracking algorithm. Thrashing can be limited by inference techniques, which will be discussed in more detail later, or by using methods that somehow “remember” the reason for the failure. Alternatives to the chronological backtrack that have been presented in the literature over the years include: dependency-directed backtracking [188], backjumping [82], intelligent backtracking [45].

Also noteworthy is the method proposed by Stallman and Sussman based on *nogoods* [188]. Nogoods are nothing more than new constraints that the search algorithm is able to add whenever a failure occurs. Nogoods have the task of “describing” the failure and preventing it from occurring again for the same assignments of values to variables. The recorded nogoods constraints can also be checked and propagated during inference to reduce the search space. Managing the discovering, recording, updating and deletion of nogoods is a non-trivial problem that has been addressed in various works [115, 89, 56, 57, 78, 161, 183].

Branching Strategies and Heuristics for Backtracking Algorithms

As described above, backtracking search algorithms, during the exploration of the search space, must make several choices when it comes to extending a node (and thus an instantiation). The usual way to extend a node $p = \{x_1 = a_1, \ldots, x_j = a_j\}$ which is a set of assignments, is by selecting a variable x_i and a value $a_i \in D_i$ and adding a branch to a new node $p \cup \{x_i = a_i\}$. This is one of the possible strategies, but other strategies have been proposed in the literature; in fact, each node $p = \{b_1, \ldots, b_j\}$ of the search tree is a set of so-called *branching constraints* where b_i, $1 \leq i \leq j$ is the branching constraint at level i in the search tree. Extending a node p consist of adding branches $p \cup \{b^1_{j+1}\}, \ldots, p \cup \{b^k_{j+1}\}$, for some branching constraint b^i_{j+1} where $1 \leq i \leq k$. In order to guarantee the completeness of the algorithm it is necessary that all branching constraints of a node are exhaustive. If we also want to ensure that no repeated solutions are explored, then branching constraints must also be mutually exclusive.

Commonly, the branching constraints are unary constraints as is the case of the three most popular branching strategies: enumeration, binary choice points and domain splitting.

In the *enumeration* strategy a variable is assigned in turn to each value in its domain, so a branch is generated for each value in its domain. In the *binary choice point* strategy for each value a_i in the domain of a variable x_i two branches are generated respectively with constraints $x_i = a_i$ and $x_i \neq a_i$. In the *domain splitting* strategy, as the name suggest, the variable is not assigned, while in each branch the possible choices for its value are reduced (e.g., $x_i \leq k$ in one branch and $x_i > k$ in the other).

When dealing with unary branching constraints, the *variable ordering heuristics*, which is used to select the next variable to branch, and the *value ordering heuristics*, which determines the order in which branches will be explored, are of fundamental importance. Various works in the literature show that these two heuristics play a crucial role in the efficient resolution of CSPs [90, 100, 86, 20].

A variable or value ordering is *optimal* if the search visits the fewest number of nodes over all possible orderings when finding one solution or showing that a solution does not exist. It is important to note that identifying an optimal variable or value ordering is at least as difficult as proving that the CSP has a solution [131], so all ordering heuristics have no formal guarantees.

The ordering heuristics that offer the best performance are those that have been specifically designed for a certain application, but there are also application-independent heuristics that we will briefly illustrate. A noteworthy case are heuristics that can adapt to the problem [147] or that are learned from previous experience in solving other instances of the same problem [67]. The search with chronological backtracking algorithm is sketched in Algorithm 1.

Algorithm 1 A simple implementation of the chronological backtracking algorithm. This implementation is based on a depth-first exploration of the search space. The function SELECT-UNASSIGNED-VARIABLE implements the *Variable Ordering Heuristics* while the function SELECT-DOMAIN-VALUE implements the *Value Ordering Heuristics*. The CONSISTENT function checks for the consistency of the current instantiation with the new assignment.

```
Require: $\mathscr{P}$ is a CSP.
 1: function SEARCH-BACKTRACKING($\mathscr{P}$)
 2:     $\mathcal{I} \leftarrow ()$                                  ▷ Current instantiation
 3:     return BACKTRACK($\mathscr{P}$, $\mathcal{I}$)
 4: function BACKTRACK($\mathscr{P}$, $\mathcal{I}$)
 5:     if $\mathcal{I}$ is complete then
 6:         return $\mathcal{I}$                                 ▷ Solution found
 7:     $x_i \leftarrow$ SELECT-UNASSIGNED-VARIABLE($\mathscr{P}$, $\mathcal{I}$)
 8:     for each $a_i \leftarrow$ SELECT-DOMAIN-VALUE($x_i$) do
 9:         if CONSISTENT($\mathcal{I}$, $\langle x_i = a_i \rangle$) then
10:             $\mathcal{I}' \leftarrow \mathcal{I} \cup \{x_i = a_i\}$
11:             $\mathcal{R} \leftarrow$ BACKTRACK($\mathscr{P}$, $\mathcal{I}'$)
12:             if $\mathcal{R} \neq fail$ then
13:                 return $\mathcal{R}$
14:     return $fail$                                    ▷ No solution
```

Variable Ordering Heuristics The variable ordering heuristic is used to select the next variable to be branched on during the search. Most of the application-independent variable ordering heuristics presented and studied in the literature are based on the size of the domains of unassigned variables. Some examples of variable ordering heuristic are: select first the variable with the smallest (largest) domain size [93], select first the variable involved in the largest number of constraints [83], select first the variable with the smallest (largest) value in the domain.

Value Ordering Heuristics When the variable ordering heuristic chooses the next variable to be branched on, the task of the value ordering heuristic is to choose the order in which the branches should be explored. Some examples of value ordering heuristic are: select value in increasing (decreasing) order, select first the value that maximizes the sum (product) of the remaining domain sizes [91, 79, 83].

Branch and Bound for Constraint Optimization Problems

Search algorithms can also be used to solve Constraint Optimization Problems. For the sake of simplicity, we will refer in the following to minimization problems, but same reasoning applies, with some modifications, to maximisation problems.

The naivest way of solving a COP would be to use the backtracking algorithm but instead of stopping at the first solution found continue the search for all possible solutions. Whenever a solution is found it can be compared with the best solution found so far and if it is better assigned as the new best. Better performance can be achieved if the cost function is exploited during the search, and this is exactly what is done by the *branch-and-bound* algorithm.

The branch-and-bound algorithm is the basic algorithm that extends the backtracking search for solving COPs. During the search, the branch-and-bound algorithm keeps the best solution found, and its cost, and each time a non-complete instantiation is proved to be more expensive than the current best solution, it backtracks, avoiding further refinements and thus reducing the search space.

Recalling that in a COP we denote with f the function that associates to each instantiation $\mathcal{I}$ of the variables of the problem a numerical value (see Section 2.1.1); we will write $f(\mathcal{I})$ to indicate that the function f is computed on $\mathcal{I}$. Furthermore, for simplicity, we can assume that if the instantiation $\mathcal{I}$ is not complete, the function f returns a lower bound of the cost of $\mathcal{I}$; that is, a value always less than or equal $f(\mathcal{I}^c)$ where $\mathcal{I}^c$ is an instantiation that extends and completes the instantiation $\mathcal{I}$.

For example, given a non-complete instantiation $\mathcal{I}$, if the result of $f(\mathcal{I})$ is already higher than the cost of the best solution found so far, the instantiation $\mathcal{I}$ can be aborted, and the algorithm performs a backtrack.

A sketch of the implementation of the branch-and-bound algorithm as an extension of the chronological backtracking algorithm is proposed in Algorithm 2.

2.1.3 Inference

Inference, which in the context of constraint programming is called *constraint propagation*, removes from the domains of variables those values that are not part of any solution (this action is also referred as *domain filtering*), eliminating large subspaces of the search space Ω. Constraint propagation is therefore an excellent method for reducing the thrashing behaviour of the backtracking algorithm introduced in the previous section.

Algorithm 2 A simple implementation of the branch-and-bound optimization algorithm, for a minimization problem, based on chronological backtracking algorithm.

```
Require: $\langle \mathscr{P}, f \rangle$ is a COP.
 1: function SEARCH-BRANCH-AND-BOUND($\langle \mathscr{P}, f \rangle$)
 2:     $U \leftarrow +\infty$                    ▷ Current upper bound (global variable)
 3:     $\mathcal{I}^* \leftarrow ()$              ▷ Best solution found so far (global variable)
 4:     BRANCH-AND-BOUND($\langle \mathscr{P}, f \rangle$, $()$, $U$)
 5:     return $\mathcal{I}^*$
 6: function BRANCH-AND-BOUND($\langle \mathscr{P}, f \rangle$, $\mathcal{I}$, $U$)
 7:     if $\mathcal{I}$ is complete then
 8:         if $f(\mathcal{I}) < U$ then
 9:             $U \leftarrow f(\mathcal{I})$
10:             $\mathcal{I}^* \leftarrow \mathcal{I}$
11:     else if $f(\mathcal{I}) < U$ then
12:         $x_i \leftarrow$ SELECT-UNASSIGNED-VARIABLE($\mathscr{P}$, $\mathcal{I}$)
13:         for each $a_i \leftarrow$ SELECT-DOMAIN-VALUE($x_i$) do
14:             if CONSISTENT($\mathcal{I}$, $\langle x_i = a_i \rangle$) then
15:                 $\mathcal{I}' \leftarrow \mathcal{I} \cup \{x_i = a_i\}$
16:                 BRANCH-AND-BOUND($\mathscr{P}$, $\mathcal{I}'$, $U$)
```

Constraint propagation can be alternated with the search phase, and/or it can be performed as an initial processing step, before the search begins. The level of local consistency achieved through constraint propagation can vary from 1 to n, but usually a higher level of consistency corresponds to a higher computational cost.

Node consistency Node consistency states that a single variable x_i is node consistent if all values in its domain D_i satisfy its unary constraints $\langle R_i, \{x_i\} \rangle$. In other words, the variable x_i with domain D_i is node consistent if and only if $D_i \subseteq R_i$. If a variable is not node consistent, it can be made so by computing: $D_i \leftarrow D_i \cap R_i$. Node consistency consists of a simple reduction of domains, once the node consistency algorithm is executed all unary constraints are satisfied. For example, given a variable x_i with domain $D_i = \{1, 2, 3, 4, 5\}$ and constraint $x_i \leqslant 3$, node consistency restricts the domain to $D_i = \{1, 2, 3\}$ and the constraint is satisfied.

Arc Consistency Arc consistency states that a variable x_i is arc consistent with respect to another variable x_j if for every value in the current domain D_i there is a value in the domain D_j that satisfies the binary constraints $\langle R_{ij}, \{x_i, x_j\} \rangle$. A set $\{x_i, x_j\}$ is arc consistent if and only if x_i is arc consistent relative to x_j and x_j is arc consistent relative to x_i. We consider the arcs $\langle i, j \rangle$ and $\langle j, i \rangle$ separately. In relational algebra an arc $\langle i, j \rangle$ is arc consistent if and only if $D_i \subset \pi_i(R_{ij} \bowtie D_j)$ where π is the projection operator. The arc $\langle i, j \rangle$ can be made arc consistent by updating the domain of the variable x_i: $D_i \leftarrow D_i \cap \pi_i(R_{ij} \bowtie D_j)$. Similar reasoning can be applied to the arc $\langle j, i \rangle$. For example, consider the binary constraint $x_i < x_j$ where the variables x_i and x_j have domain $D_i = D_j = \{1, 2, 3\}$. For the value $\{3\}$ in d_i there is no value in d_j that can satisfy the constraint and therefore the value $\{3\}$ can be removed from D_i. The same reasoning can be applied for the value $\{1\}$ in D_j so it can be removed.

If during the consistency check all the arcs are already arc consistent then the CSP is arc consistent otherwise if one domain is changed the algorithm needs to recheck all constraints whose variables have experienced a domain change.

The most widely used arc consistency algorithm is AC-3 presented by Mackworth in 1977 [140]. Assuming we have a CSP with n decision variables, each with domain size no greater than d and with a number m of binary constraints, the worst-case running time of the AC-3 algorithm is $O(md^3)$.

Path consistency and k-consistency Path consistency [149] considers triplets of variables. A set of two variables $\{x_i, x_j\}$ is path consistent with respect to a third variable x_m if, for each assignment consistent with the binary constraints $\langle R_{ij}, \{x_i, x_j\} \rangle$, there exists an assignment for x_m that satisfies the binary constraints $\langle R_{im}, \{x_i, x_m\} \rangle$ and $\langle R_{mj}, \{x_m, x_j\} \rangle$. A path of length two from node i through node m to node j is path consistent if and only if $R_{ij} \subset \pi_{ij}(R_{im} \bowtie D_m \bowtie R_{mj})$ and can be made path consistent by computing: $R_{ij} \leftarrow R_{ij} \cap \pi_{ij}(R_{im} \bowtie D_m \bowtie R_{mj})$. A set of three variables $\{x_i, x_j, x_m\}$ is path consistent if and only if for any permutation of (i, j, m), R_{ij} is path consistent relative to x_m.

Further levels of consistency fall under the concept of k-consistency introduced by Freuder in 1978 [76]. Note that within this concept 1-consistency is equivalent to node consistency, 2-consistency to arc consistency and 3-consistency to path consistency. k-consistency requires that given consistent values for any $k - 1$ variables, there exists a value for any k-th variable, such that all k values are consistent. The k-consistency is not sufficient to guarantee the satisfiability of a CSP if it has more than k variables.

Higher consistency levels can generally do more propagation, but are also more computationally expensive, so arc consistency is often the most widely used method because it has the best compromise between the propagation performed and the computational cost.

Generalized Arc Consistency (GAC) The consistency algorithms mentioned so far only refer to unary or binary constraints; in case some of the CSP constraints are p-ary ($p > 2$), as in the case of global constraints, it is possible to generalize the concept of arc consistency.

A constraint $\langle R_{S_j}, S_j \rangle$, where $S_j = \{x_1, \ldots, x_k\}$, is Generalized Arc Consistent if and only if for each variable $x_i \in S_j$, for each value $a \in D_i$ there exists an assignment of the remaining $n - 1$ variables $x_1, ..., x_{i-1}, x_{i+1}, ..., x_k$ such that the constraint is satisfied [148]. Each p-ary constraint is represented as an hyperarc connecting the vertices representing the variables in S_j. Given a directional hyperarc $\langle x_i, S_j - \langle x_i \rangle \rangle$ this is made generalized arc consistent by updating the domain D_i according to the following formula: $D_i \leftarrow D_i \cap \pi_i(R_{S_j} \bowtie (\bowtie_{m \in S_j - \langle x_i \rangle} D_m))$.

The computational complexity of enforcing Generalized Arc Consistency (GAC) strongly depends on the constraint being implemented. Enforcing GAC in general is an NP-COMPLETE problem; however, there are cases where particularly efficient implementations manage to enforce the GAC with polynomial complexity: probably the most famous case is that of the `alldifferent` constraint [170].

Consistency levels are just a convenient way to indicate how “strong” the constraint propagation is. In practice, the goal is not to achieve a specific consistency level but to implement constraints in such a way that that have a good trade-off between performed propagation and computational cost.

2.2 Implementing Constraint Propagation

2.2.1 Propagators

In constraint programming systems the constraints expressing the existing relations between the decision variables are implemented by means of so-called *propagators*. In this section we will briefly present the structure of a propagator and the structure of the propagation engine, i.e. the algorithm that coordinates the execution of the various propagators in order to perform the constraint propagation.

The simplest way to represent a constraint may be through an extended definition listing all possible assignments for the involved variables. In constraint programming systems, constraints are rarely handled through an extended representation, mainly for efficiency reasons. Representing all the possible assignments of a constraint would require an exponential amount of memory in the number of variables, and the possibility of exploiting particular structures would be lost.

Constraint programming systems *implement* each constraint by a collection of *propagators* (also known as *filter*). The task of a propagator is to “observe” the domains of the variables involved in the constraint and, as soon as a value is removed from the domain of a variable, try to remove further values from the domains of its variables. The algorithm used by a propagator is often referred as *filtering algorithm*. In the following, we write $p(D_i)$ to indicate that the propagator p is executed on the domain D_i, we also write $p(\mathcal{D}_i)$ to indicate that a propagator p is executed on a set of domains $\mathcal{D}_i$. The *input* variables for a propagator are the variables used to perform the computations (the variables used by the filtering algorithm) while the *output* variables are the variables whose domain is modified by the propagator. It is usual for input and output variables to coincide.

Definition 2.2.1. (Propagators)

- A propagator must be *contracting* (or decreasing), in order to guarantee that constraint propagation only removes values: $\forall D_i \in \mathcal{D}, p(D_i) \subseteq D_i$
- A set of *monotonic* propagators guarantee that the order in which propagators are applied does not change the result: $\forall D_i, D_j \in \mathcal{D}, D_i \subseteq D_j \Rightarrow p(D_i) \subseteq p(D_j)$.
- A propagator p is at *fixpoint* on a domain D_i if and only if applying p to D_i gives no further propagation: $p(D_i) = D_i$
- A propagator p is *idempotent* if the result of a propagation is a fixpoint of p: $\forall D_i \in \mathcal{D}, p(p(D_i)) = p(D_i)$
- A propagator p is *correct* for a constraint c if and only if it does not remove any assignment that is consistent with c.
- A propagator p is *entailed* by a set of domains $\mathcal{D}_i$, if all set of domains $\mathcal{D}_j$ with $\mathcal{D}_j \sqsubseteq \mathcal{D}_i$ are fixpoints of p.

To solve a CSP, a constraint programming system must implement both strategies presented in the previous sections: search and constraint propagation. The search, carried out by the module known as the *search engine*, is implemented via a search procedure e.g., the backtracking search described in Section 2.1.2. The propagation, instead, is carried out by the *propagation engine* represented by the procedure PROPAGATE sketched in Algorithm 3. Note that the procedure presented is straightforward, various optimizations are possible, the most common being early detection of failures and entailments. The SELECT-PROPAGATOR procedure, which is responsible for selecting the order of application of the propagators, will be discussed in the following section.

2.2.2 Life Cycle of a Propagator

The life cycle of a propagator can be summarised in the following status: subscripted, runnable, executed, suspended; and is illustrated in Figure 2.1.

Subscripted. When a propagator p is created, it *subscribes* to its input variables. The subscription allows the propagation engine to run the propagator whenever the domain of one of its variables changes.

How a domain changes is described by *propagation events* or just *events*. The usual events defined in a constraint programming system are:

- fix(x): the variable x become fixed.
- minc(x): the minimum value in the domain of variable x changes.

Algorithm 3 Sketch of a basic propagation engine for a constraint programming system. In line 8 the set M is composed of the variables whose domain has changed after the execution of the propagator p. In line 9, input(p') represents the input variables of the propagator p'.

```
Require: $P_f$ set of propagators at fixpoint for $\mathcal{D}_i$.
Require: $P_n$ set of propagators not known to be at fixpoint for $\mathcal{D}_i$.
Require: $\mathcal{D}_i$ a set of domains.
 1: function PROPAGATE($P_f$, $P_n$, $\mathcal{D}_i$)
 2:     $N \leftarrow P_n$
 3:     $P \leftarrow P_f \cup P_n$
 4:     while $N \neq \emptyset$ do
 5:         $p \leftarrow$ SELECT-PROPAGATOR(N)
 6:         $N \leftarrow N - \{p\}$
 7:         $\mathcal{D}_j \leftarrow p(\mathcal{D}_i)$                          ▷ Execute the propagator $p$
 8:         $M \leftarrow \{x \in X \mid \mathcal{D}_i(x) \neq \mathcal{D}_j(x)\}$     ▷ $M$ is the set of modified variables
 9:         $N \leftarrow N \cup \{p' \in P \mid \text{input}(p') \cap M \neq \emptyset\}$
10:         if $p$ is idempotent then
11:             $N \leftarrow N - \{p\}$
12:         $\mathcal{D}_i \leftarrow \mathcal{D}_j$
13:     return $\mathcal{D}_i$
```

- maxc(x): the maximum value in the domain of variable x changes.
- any(x): the domain of the variable x changes.

Runnable. After the subscription the propagator waits in the runnable set until it is selected by the propagation engine for the execution. The order in which propagators are selected to be applied, is performed by the function SELECT-PROPAGATOR in line 5 of Algorithm 3. Each system in this case uses its own selection policy, but normally priority is given to propagators with less complexity or resumed by events that provide more information (e.g. fix). Priorities can be static or dynamic. The ECL^{i}PS^e constraint programming system [184], which will be mentioned in the following, for example, supports 12 different priority levels.

Executed. A propagator that is selected for execution is said to be *resumed*, *awakened*, or *activated*. The execution of p can have one of three possible outcomes.

In the first case the propagator p realizes that the constraint has no solution, e.g., because a domain becomes empty as a result of deleting some values. In this case the propagator is deallocated, and a failure is returned.

In the second case, the propagator finds that the constraint is entailed i.e., it is satisfied whatever the values assumed by the variables among those remaining in the domains. Also, in this case the propagator is deallocated.

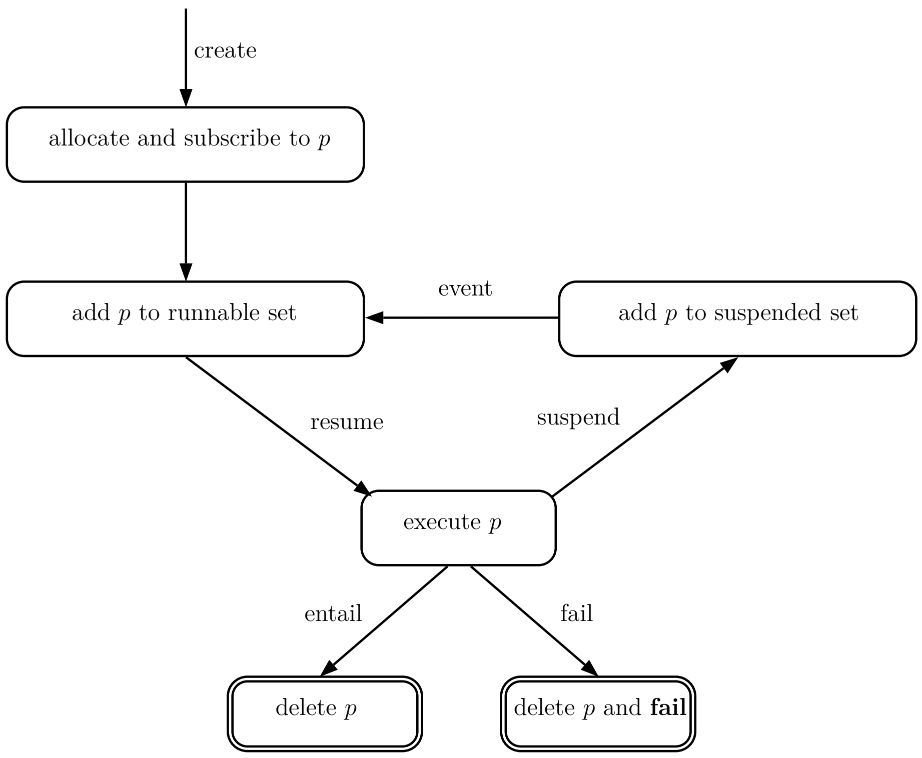


Figure 2.1: A sketch of the life cycle of a propagator p in a constraint programming system.

In the third case, neither of the previous two cases has occurred and so the propagator p at the end of the execution, when it reaches the fixpoint, is moved to the set of suspended propagators waiting for an event to occur.

Suspended. A propagator that has reached the fixpoint waits in the suspended set until the domains of the involved variables change. These domains change triggers an event, and the propagator is moved to the runnable set (adding the propagator p to the runnable set is called *scheduling* p).

2.3 Constraint Logic Programming

Constraint Logic Programming is the fusion of two declarative paradigms: constraint programming and logic programming. The term was coined by Jaffar and Lassez in 1986 [108] when they provided a first schema and semantics for CLP languages. In CLP it is possible to keep the logical formulation of the problem separate from the solving algorithm, making the programming languages used in this area much more expressive than other existing approaches for solving constraint problems.

2.3.1 Logic Programming

Logic programming is a declarative programming paradigm in which programs are not made up of statements or functions, as in imperative or functional programming, but are based on formal logic. A logic program consists of a set of rules written in the form of *clauses*.

Definition 2.3.1. A *literal* is an atomic formula (atom) or its negation. Each literal involves an n-ary predicate p, and has the form $p(t_1, \ldots, t_n)$ where t_i is called a *term*. A *term* is recursively defined and can be: a variable x, a constant c or an n-ary function f applied to n terms: $f(t_1, \ldots, t_n)$.

A *clause* is an expression that has the form $H :- B$. H is called the *head* of the clause while $B = B_1, \ldots, B_m$ is called the *body* and is a conjunction of literals. A clause is true when all its literals are true. The symbol $:-$ is an implication from the body B to the head H: if B is true, then H must be true. When a clause does not have a body, it is called a *fact* and is written $H :- \Box$, where $\Box$ denote the empty sequence, or in the simplified form: H.

If we want to see a logic program from the perspective of classical imperative programming, the head of a clause is the definition of a procedure (the predicate) with its arguments (the terms) and the body is its code.

The execution of a logic program corresponds to asking for the truth value for a certain statement $:- G$ which is called the *goal*. Asking for the truth value of G corresponds to asking whether there are assignments to variables in G such that G is true given the clauses of the logic program ($G = G_1, \ldots, G_n$ is a conjunction of literals).

This is answered by repeatedly transforming the goal through a series of *resolution steps*, until: the empty goal is reached (success), the resolution cannot be continued and the goal is not empty (failure) or the resolution continues indefinitely.

Each resolution step involves *unifying* a literal, that is part of the goal, with the head of a clause.

Definition 2.3.2. (Substitution) Let $\mathcal{V}$ be a set of variables and let $\mathcal{T}$ be a set of terms. A substitution is a function $\sigma : \mathcal{V} \to \mathcal{T}$. Substitutions are traditionally written as postfix operators. Given a term t, $t\sigma$ is defined recursively as follows:

- if c is a constant, $c\sigma = c$;
- if x is a variable, $x\sigma = \sigma(x)$;
- if f is a n-ary function, then $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$.

Definition 2.3.3. (Unification) Two or more literals can be *unified* if there exists a substitution σ of terms for the variables of the literals that makes them (or more precisely their instances) equal. The substitution σ is called *unifying substitution* and is written:

$$[G_1]\sigma = [G_2]\sigma = \cdots = [G_n]\sigma$$

In general, there can be several substitutions and we want to identify the most general one, called *most general unifier* (*mgu*).

Let θ and σ be two substitutions, θ is said to be *more general* then σ (written $\theta \leq \sigma$) if there exists a substitution η such that $\sigma = \theta\eta$.

Given two literals s and t the substitution θ is an *mgu* (most general unifier) if θ is a unifier and for every unifier σ of s and t it holds that $\theta \leq \sigma$.

If the unification of the literal A that is part of a goal $G = A, R$ is successful with the clause $H :\!- B$, i.e. the most general substitution σ has been found, then the current goal can be replaced by the new goal $(B, R)\sigma$. That is, we replace A with the body of the clause, and we apply σ to the whole new goal. More precisely, we can write:

$$\frac{G = A_1, \dots, A_n \quad H :\!- B_1, \dots, B_m \quad \exists\sigma : [A_i]\sigma = [H]\sigma}{G' = [A_1, \dots, A_{i-1}, B_1, \dots, B_m, A_{i+1}, \dots, A_n]\sigma}$$

A sequence of resolution steps is called a *derivation.* If a derivation ends with success is called a *refutation.* The set of all derivations starting from a goal can be represented as a tree, called derivation tree, with the root being the goal and the leaves being the empty goal (success nodes) or failure nodes.

2.3.2 Adding Constraints to Logic Programming

Running a logic program can be considered as an exploration of the derivation tree. Prolog, which is one of the best-known logic programming languages, uses a depth-first exploration very similar to the backtracking algorithm described in Section 2.1.2. It is therefore easy to identify a logic language as a good starting point for modelling constraint satisfaction problems on finite domains.

Constraints are nothing more than relations that can be modelled, in logic programming, by sets of facts where the predicate name corresponds to the name of the constraint. Whether all the constraints must be satisfied can be implemented by a clause whose body contains all the constraints predicates while the head contains all the variables in the constraints predicates; asking for a goal that corresponds to the head of the clause corresponds to asking for a solution for the CSP.

The execution engine of a logic program can find a solution of the CSP defined in this way, but the fact that it bases the resolution exclusively on an exploration of the derivation tree makes this resolution method particularly inefficient since in CSPs the search space is often very large.

The Constraint Logic Programming (CLP) paradigm involves extending the logic execution engine with dedicated constraint solvers. Each constraint solver, depending on the class of constraints it can handle, yields a *constraint language*, e.g.: CLP(FD) for finite domains, CLP(R) for real domains, CLP(Bool) for Boolean domains.

Finite domains are one of the fields in which CLP has been most successful. Historically, the first CLP(FD) language was CHIP [60], developed since 1985. While the original semantics for CLP were limited to solving constraint satisfaction problems, modern languages provide optimisation mechanisms to solve COPs as well.

Given a constraint language, a CLP clause is just a logic programming clause except that its body contains in addition constraints of considered language. When a resolution step is performed on a CLP clause it is no longer only necessary to verify the existence of a *mgu* between the selected subgoal and the head of a clause, but it is also necessary to verify the consistency of the current set of constraints with the constraints in the body of the clause.

If in logic programming a state consists of a goal and a substitution, in CLP it consists of a goal and a set of constraints called the *constraint store*.

Definition 2.3.4. (Resolution in CLP) Let $\langle G, S\rangle$ be a state where $G = A, R$ is the current goal and S is the current constraint store. Assume that we want to replace A, then:

- If A is a constraint: A is added to S and the new state is $\langle R, prop(S \wedge A)\rangle$, where $prop(C)$ is the result of applying some constraint propagation algorithm to the constraint store C.
- If A is a literal: if there is a clause $H :- B$ with the same head predicate as A, the constraint $A = H$ is added to the constraint store and A is replaced with B. The new status results accordingly $\langle (B, R), S \wedge \{A = H\}\rangle$.

A computation of a CLP is *successful* if there is a derivation from the initial state $\langle G, \emptyset\rangle$ to the state $\langle G', S\rangle$ where G' is the empty goal and S is satisfiable.

If in any resolution step the consistency of the constraint store does not hold, a failure occurs and a backtrack is performed. In this way the constraints are exploited for the early detection of failing, increasing the performance in CSPs solving.

Constraint Logic Programming Languages

The first CLP language to be developed was Prolog II in 1972 [49], a few years later Jaffar and Lassez provided a first schema and semantics for CLP languages [108]. Since then, other CLP languages have been developed in particular: Prolog III in

1990 [50], CHIP in 1988 for constraints on finite domains [60] and CLP(R) in 1992 with support for arithmetic constraints on floating point numbers [110].

More recently, hybrid solving techniques that combine constraint propagation with linear programming (or mixed-integer programming) typical of operations research have emerged. The CLP language ECL^{i}PS^e [184] falls within this type of approach, and this enabled it to be used also in modern industrial-scale applications.

ECL^{i}PS^e is a programming system, based on the Prolog logic programming language, used to model CSPs and COPs in a declarative manner. In the same declarative way, it also allows the development of new constraints and their related filtering algorithms but also new customised heuristics to extend backtracking search algorithm.

The fact that it is based on a declarative language, in addition to its overall performance, makes it an excellent tool for designing and developing of solving techniques, especially in the research field. The system is also distributed as open source since 2006. For this reason, all the algorithms proposed in the following chapters of this thesis have been implemented with it.

3 The Euclidean Traveling Salesperson Problem

Contents

The Traveling Salesperson Problem (TSP) is one of the best-known COP in computer science. The Euclidean Traveling Salesperson Problem (ETSP) is a special case in which each node is identified by its coordinates on the plane and the Euclidean distance is used as cost function.

No specialized pruning algorithm have been proposed in CP for the Euclidean TSP, and the usual way to tackle Euclidean TSPs is to compute the distance matrix and address the problem as a general TSP. It is worth noting that in the Euclidean TSP more information is available than in the general TSP: the coordinates of the points to be visited are known, and geometrical concepts (straight line segments, angles, etc.) can be defined in the Euclidean plane. In this chapter we present the first attempt (to the best of our knowledge) to exploit the geometric information in order to achieve a stronger pruning in CP when solving Euclidean TSPs. We address the pure Euclidean TSP (without side constraints) and propose new constraints that efficiently reduce the search space. We show that the pruning we introduced is not subsumed by that of important works in the area [27].

We also conducted a study on the performance of the newly introduced constraints and found that not all of them are equally useful. Hence, it is important to define a way of classifying useful constraints. To do this, we use machine learning approaches with the aim of only imposing those constraints that have been classified as effective. We compare two classifiers based on Random Forest and Neural Networks that are found to be effective.

3.1 Preliminaries

3.1.1 The Traveling Salesperson Problem (TSP)

Given a weighted graph the Traveling Salesperson Problem (TSP) requires to compute the minimum cost cycle that visits each vertex exactly once. A cycle that visits each vertex exactly once is often referred as *Hamiltonian cycle* or *Hamiltonian circuit*. The name *Traveling Salesperson Problem* comes from the problem's most famous formulation: *"A salesman has to visit a set of cities, each of which must be visited only once, and he wants to minimize the length of the tour"*.

History

The origin of the name of the problem and its formulation are not clear. To date, no official documentation is known that contains the name of the creator, nor is there any speculation about its first use [14]. Traces of the problem seem to go back to a German handbook for street vendors from 1832, the problem was then addressed mathematically by Willian Rowan Hamilton and Thomas Penyngton Kirkman later in the same century. The first document to include the term is a 1949 publication by Julia Robinson "On the Hamiltonian game (a traveling salesman problem)" [177], but the contents indicate that she did not introduce the name of the problem.

The current formulation of the TSP appears to have been included in the literature for the first time by Karl Menger during a mathematicians colloquium in Vienna on February 5, 1930 [145]; later it was the *cutting planes* method introduced in 1954 by George Dantzig, Delbert Ray Fulkerson and Selmer Martin Johnson that provided the first significant result in solving the problem [19]. The cutting planes method was the first to model the problem as an Integer Linear Programming (ILP) problem, a representation that is still used in the most efficient solvers available today.

Despite its simple formulation the TSP is difficult to solve, the evidence of which dates to 1972, when Richard M. Karp first showed that the problem belongs to the NP-HARD complexity class [114]. It was not until the 1990s that the team of David K. Applegate, Robert E. Bixby, Vaclav Chvatak and Willian J. Cook developed the *Concorde* solver [13]. Concorde, based on ILP technologies, is still considered in practice to be the best algorithm for solving TSPs.

TSP solving strategies

Excluding the trivial computation of all permutations looking for the one with the lower cost (complexity $\mathcal{O}(n!)$), which becomes impractical even for very small values of n, the TSP solving strategies presented in the literature are many.

A first classification can be made according to the type of solution generated; solving strategy are thus divided into two classes: heuristic algorithms and exact algorithms.

Heuristic algorithms are not always able to find the true optimal solution, but they are nevertheless able to find good quality solutions in a reasonable time. Heuristic algorithms are further divided into other subcategories: constructive heuristics and improvement heuristics. A constructive heuristic iteratively updates the solution (starting with an empty solution), each time trying to insert the most promising element among all the admissible ones, until the solution is complete. Most common constructive heuristics for the TSP are: Nearest Neighbor [121], First Fit, First Fit Decreasing, Strongest Fit, Strongest Fit Decreasing, Cheapest Insertion, Regret Insertion, Christofides algorithm [48].

Improvement heuristics, on the other hand, start from an admissible solution of the problem and iteratively apply a series of partial modifications with the aim of finding a better solution. The best-known improvement heuristics for TSP are: 2-opt [53], 3-opt, k-opt, Lin-Kernighan [136], Tabu-Search, Simulated Annealing [135], Genetic Algorithms [92, 40].

Exact algorithms, on the other hand, as the name suggests, can find the optimal solution to the problem. Among the exact solving algorithms, as also mentioned in Section 3.1.1, Concorde is still considered to be the most efficient one available in most applications. Concorde is based on ILP techniques such as branch-and-cut and is able to optimally solve even instances with tens of thousands of instances [13].

However, most of the algorithms developed for solving the TSP, including Concorde, can only be used when the problem is in its original formulation, whereas when the TSP has to be solved in problems that impose additional constraints, it becomes necessary to adopt alternative solving techniques. Among the alternative approaches, CP based approaches, which will be presented in this thesis, are certainly noteworthy, given their flexibility in handling variants of the problem.

3.1.2 Variations and special cases of TSP: The ETSP

Due to its wide notoriety, many variants of the TSP and some special cases have been studied in the literature over the years; variants partially modify the problem formulation, often introducing additional constraints, while the special cases relate to the fact that additional assumptions are imposed on the available data.

The TSP with Multiple Visits (TSPM) relaxes the degree constraints of the nodes, since each node in the graph no longer has to be visited exactly once, but at least once [4]. The Clustered TSP introduces a partition of the nodes of the graph $V_1, V_2, \ldots, V_k$. The problem requires finding the minimal path in which the nodes within each cluster are visited consecutively [99, 112]. The Ordered Cluster TSP is a simplified version in which the clusters must be visited in order, i.e., all nodes in cluster V_1 are visited first, followed by all nodes in cluster V_2 and so on [11]. The Generalized TSP (GTSP) implies, as in the previous cases, a division of the nodes of the graph, but in this case, it is necessary to find the minimum path that goes through exactly one node of each cluster [70, 126]. In the Resource Constrained TSP (RCTSP) in each edge of the graph a requirement r_{ij} is added in addition to the cost; solving the problem involves finding a minimum cost tour in which the sum of the requirements does not exceed a given constant R [157]. The Precedence Constrained TSP introduces precedence constraints of type $B_{ij} = \{(i,j) | i \in V, j \in V, i \neq j\}$ which state that the node i must be visited before node j, then is required to identify the minimum path that meets all constraints [21]. In the TSP with Time Windows (TSPTW) every edge $e_{i,j}$ of the graph is associated with a traversal time t_{ij} in addition to the cost. Each node has an associated time window $[a_i, b_i]$ representing the time frame during which the service at node i must be executed. Also, each node has an associated duration s_i, which represents the duration of the service to be performed at node i (with $a_i \leq b_i$ and $0 \leq s_i \leq b_i - a_i$). If node i is reached before time a_i it is necessary to wait until a_i to perform the service [158, 17, 18].

Metric TSP and Euclidean TSP

With regard to special cases, the *metric TSP* and the *Euclidean TSP* are known in the literature. Many benchmarks for the TSP, see for example the TSPLIB [172], relate precisely to these cases. In the metric TSP (sometimes referred as delta-TSP), the costs associated with each arc, and thus the distance between cities, satisfy the triangle inequality: the direct route between two cities is never longer than the route through an intermediate city. In the Euclidean TSP, the used metric is the Euclidean distance: given two points $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$

the Euclidean distance in $\mathbb{R}^2$ is calculated as $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Since the Euclidean distance respects the triangular inequality, the Euclidean TSP is a special case of the metric TSP. In the Euclidean TSP, more data is available on the problem; in particular, the coordinates in the plane of the various cities to be visited are known. The Euclidean variant of TSP has been shown to maintain the same complexity (NP-HARD) as the more general case [81].

In 1996 Sanjeev Arora showed that the Euclidean TSP admits a polynomial-time approximation scheme (PTAS) [16]. A PTAS is an algorithm that, given an instance of an optimization problem and fixed a parameter $\varepsilon > 0$, produces in polynomial time a solution within a factor $1 + \varepsilon$ of the optimum. Note that the computation time can still be an exponential function of $\frac{1}{\epsilon}$. In the case of TSP this corresponds to saying that the algorithm would be able to find in polynomial time a path of length at most $(1 + \varepsilon)L^*$, with L^* length of the optimal path. The PTAS algorithm developed by Arora has a spatial and temporal complexity of $\mathcal{O}(n^{2d}(\log n)^{\mathcal{O}((\frac{\sqrt{d}}{\varepsilon})^{d-1})})$ in $\mathbb{R}^d$ (which reduces to $\mathcal{O}(n(\log n)^{\mathcal{O}((\frac{\sqrt{d}}{\varepsilon})^{d-1})})$ in $\mathbb{R}^d$ if a quad-tree is used [181]).

3.1.3 Notation and definitions

Let $G = (V, E, w)$ be a weighted graph, where V is a set of nodes ($|V| = n$), E is a set of edges, and $w : E \mapsto \mathbb{R}^+$. A *path* in G is a sequence $p_{v_{s_0} - v_{s_k}} = v_{s_0} e_{s_0,s_1} v_{s_1} \dots e_{s_{k-1},s_k} v_{s_k}$ such that:

(*i*) $v_{s_0}, v_{s_1}, \dots, v_{s_k} \in V$ and are all distinct, and

(*ii*) $e_{s_0,s_1}, e_{s_1,s_2}, \dots, e_{s_{k-1},s_k} \in E$.

Since a path is uniquely identified by the sequence of its nodes (or of its edges) in the proper order, we will often write paths as sequences of nodes to simplify the notation. The length of a path p is the sum of the weights of its edges: $L(p) = \sum_{i=0}^{k-1} w(e_{s_i,s_{i+1}})$. A *circuit* c is a sequence obtained by appending e_{s_k,s_0} to a path $p_{v_{s_0} - v_{s_k}}$.

In the Euclidean case, let $\mathcal{P} = \{P_1, \dots, P_n\}$ be a set of points, where $P_i = (x_i, y_i)$. The graph associated with $\mathcal{P}$ is $G^{\mathcal{P}} = (\mathcal{P}, E^{\mathcal{P}}, w^{\mathcal{P}})$, where $E^{\mathcal{P}} = \{e_{i,j} \equiv (P_i, P_j) \mid P_i, P_j \in \mathcal{P}, i \neq j\}$ and $w(P_i, P_j) = d(P_i, P_j)$, where d is the Euclidean distance.

We use $\overline{P_iP_j}$ to denote the segment in the plane with the extremes P_i and P_j. Since in the Euclidean case every edge of a graph corresponds to a segment in the plane between the corresponding endpoints, we will often confuse the edge $e_{i,j}$ with the corresponding segment $\overline{P_iP_j}$. Also, we will sometimes confuse the index $i \in V$ with the corresponding point P_i in the plane. We denote with $\overleftrightarrow{P_iP_j}$ the (infinite straight) line passing through points P_i and P_j, and with $\angle P_iP_jP_k$

the counterclockwise angle formed by the segments $\overline{P_iP_j}$ and $\overline{P_jP_k}$ with vertex in P_j from P_i to P_k.

A *crossing* in a path p is defined as a common point $P_q \notin \mathcal{P}$ that is shared by two (or more) edges of p or a commmon point $P_r \in \mathcal{P}$ that is shared by three (or more) edges of p.

3.1.4 Integer linear programming representation

Although this thesis covers TSP resolution in the CP domain, here we briefly describe the ILP representation of the problem and its constraints.

When $w(e_{i,j}) = w(e_{j,i})\ \forall(i,j) \mid i \in V, j \in V\ i \neq j$ the TSP is called *symmetric*, while the case in which the cost of an arc depends instead on the direction of travel then it is called *asymmetric* TSP (ATSP). Depending on whether the problem is symmetric or asymmetric, its representation is different. The Eculidean TSP is symmetric.

A symmetric TSP formulation can be described as follows:

$$x_{ij} = \begin{cases} 1, & \text{if edge } e_{i,j} \text{ is in the solution} \\ 0, & \text{otherwise} \end{cases}$$

$$z = \text{Minimize} \sum_{i>j} w(e_{i,j}) x_{ij} \qquad i,j = 1,\dots,n \tag{3.1}$$

$$\text{s.t.} \sum_{j} x_{ij} = 2 \qquad i = 1,\dots,n; i \neq j; x_{ij} \equiv x_{ji} \tag{3.2}$$

$$\sum_{i,j \in S} x_{ij} \leq |S| - 1 \qquad \forall S \subset V : 3 \leq |S| \leq |V| - 3 \tag{3.3}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in V, i \neq j$$

Equation (3.1) is the objective function; we want to minimize the cost of the tour.

The constraint (3.2) is called *degree constraint* and assure that, in the solution, the degree of each node is equal to two.

Let S be a subset of the set of nodes V, under the condition S have at least three elements. The constraints (3.3) enforce that there must be at least one incident arc in a node of S and in a node of $V \setminus S$ thus ensuring that all nodes will belong to the same cycle, and not disjoint tours that only collectively cover all cities. We often refer to constraints (3.3) as *subtour elimination constraints.* Given a set of n elements the number of possible subsets is equal to 2^n, it follows that there are an exponential number of subtour elimination constraints with respect to the cardinality of V (note that the subtour elimination constraint is not imposed on subsets with a number of elements less than 3 or greater than $|V| - 3$; in general, however, the number of constraints remains exponential).

3.1.5 Modelling the TSP in Constraint Programming

In the CP literature, three main representations have been devised for defining variables in the Hamiltonian circuit problem and the TSP: the *permutation* representation, the *successor* representation, and the *set variable* representation [27]. The last was also extended to the graph representation [61, 68, 69].

In the *permutation* representation, n variables Pos_i are introduced $Pos = (Pos_1, Pos_2, \ldots, Pos_n)$ each with initial domain V; variable Pos_i represents the i-th node that is visited. For example, if $n = 5$ and $Pos = (3, 5, 4, 2, 1)$ the corresponding tour will be $(3, 5, 4, 2, 1, 3)$. The constraint model for the permutation representation includes an `alldifferent`(Pos) constraint [170] on the Pos array of variables, that ensure that each node is visited only once.

The *set variable* representation is based on a set variable Set which represents the edges that form the tour [88, 163]. For the set domain representation, the most natural representation, which is also used in [27], is the definition "cardinality + subset" which requires: that Set has cardinality n and that there is a lower bound $L(Set)$ representing all *mandatory* edges (edges that are certainly part of the solution), and an upper bound $U(Set)$ representing all *possible* edges (edges that might be part of the solution).

In the *successor* representation, n variables $Next_i$ are defined $Next = (Next_1, Next_2, \ldots, Next_n)$; variable $Next_i$ represents the node that follows node i in the circuit, and its initial domain is $\{1, \ldots, n\} \setminus \{i\}$. For example, if $n = 5$ and $Next = (3, 5, 4, 2, 1)$ the corresponding tour will be $(1, 3, 4, 2, 5, 1)$.

The constraint model of the successor representation includes, as in the permutation representation, an `alldifferent`($Next$) constraint on the $Next$ array of variables, but in this case the `alldifferent`($Next$) constraint ensures that each node has exactly one incoming edge (degree constraints). The constraint model for the successor representation also includes a `circuit`($Next$) [22, 46, 117] constraint (sometimes called `nocycle`) that avoids subtours, i.e., cycles of length less than n (subtour elimination constraints).

In some cases, the constraint model includes, as redundant representation, also a set of $Prev$ variables: $Prev_i$ represents the node that precedes node i in the circuit. Often, constraint models include redundant representations to obtain additional pruning.

Both the `alldifferent` and the `circuit` constraints are already implemented in most constraint logic programming systems. Hereafter, unless otherwise specified, we refer always to the successor representation.

3.2 Solving the TSP in Constraint Programming

As already introduced, the travelling salesman problem is probably one of the best known and most treated COP in the scientific literature. Many approaches proposed for its resolution have been published over time, even if we refer only to Constraint Programming. Usually, when it comes to exploit the objective function to prune the search space, CP formulations are not as effective as Integer Programming models, but they are more flexible in dealing with side constraints. In an effort to combine the advantages of CP and ILP approaches the development of hybrid techniques comes as a matter of course.

Various works propose to use relaxations of the TSP to prune suboptimal branches; the classical relaxations of the TSP are the assignment problem and the one-tree relaxation. Caseau and Laburthe [46] propose a simple and effective rule for the `circuit` constraint (called `nocycle`), and also propose to filter values based on the objective function. For this purpose, they apply the assignment-based and the spanning tree relaxation. The rule consists in finding a path of mandatory edges of length at most $n - 1$ and removing the edge between the two endpoints of the path.

Kaya and Hooker [117] propose a new filtering rule based on separator graph, able to remove nonhamiltonian edges, for the `circuit` constraint.

Francis and Stuckey consider various propagation algorithm for `circuit` and provide for each explanation in the context of a lazy clause generation solver [75].

Fages and Lorca [68] show how properties of the reduced graphs (obtained by introducing a node for each Strongly Connected Components (SCC) of the original graphs) associated to Asymmetric TSPs can be used to improve the Minimum Spanning Tree (MST) relaxation.

Pesant *et al.* [158] address the TSP with Time Windows (TSPTW) and exploit the `circuit` constraint together with the MST relaxation. Focacci *et al.* [74, 73] propose reduced costs filtering to optimization constraints, and in particular use the assignment problem and the minimum spanning forest relaxation.

The groundbreaking work in this area is that by Benchimol *et al.* [27]: it is the first in which a CP model was able to solve large TSP instances. In their work, they use a variety of techniques. They propose an implementation of the weighted circuit constraint that includes the Held and Karp [101] scheme, iterates a Lagrangian relaxation to obtain a high-quality one-tree, and uses it to remove edges similarly to reduced costs filtering. It also identifies as mandatory those edges that, if removed, would increase the current lower bound over the quality of the incumbent solution. To find quickly a solution, they first run the Lin-Kernighan-Helsgaun algorithm [136, 102]. In the experiments with asymmetric TSPs, they also use additive bounding [71] to combine both the 1-tree and the assignment problem relaxations.

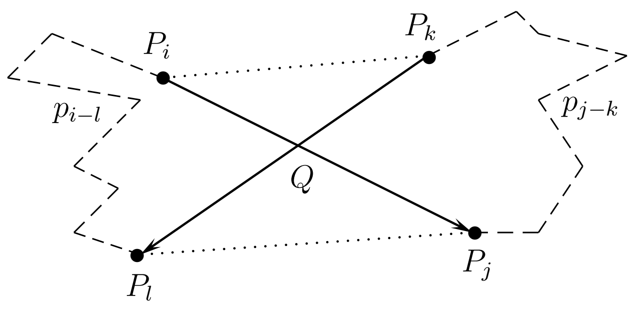


Figure 3.1: A self-crossing circuit. Theorem 3.3.1 suggests that instead of taking $\overline{P_iP_j}$ and $\overline{P_kP_l}$, a shorter tour chooses the dotted edges $\overline{P_iP_k}$ and $\overline{P_jP_l}$

In [69], the authors further improve by casting the problem in CP(Graph) and by means of improved search heuristics (i.e., Last Conflict heuristic).

Isoart and Régin [107] design a propagator based on the search of k-cutsets. The combination of this constraint with the Weighted Circuit Constraint (WCC) constraint has resulted in a significant reduction in the computation time.

3.3 Avoiding crossings

Among the geometric information that can be exploited to solve Euclidean TSPs, a well-known result in the literature is that the optimal solution of a metric TSP (thus, also of a Euclidean TSP) in the plane cannot include two edges that cross each other (see Figure 3.1).

Theorem 3.3.1 (Flood [72]). *Let c^* be an optimal tour of a metric TSP. Then, for each $e_{i,j}, e_{k,l} \in c^*$ such that $\{i, j, k, l\}$ are all different and not all aligned, the segments $\overline{P_iP_j} \cap \overline{P_kP_l} = \emptyset$.*

Proof. By contradiction, suppose that the optimal tour c^* contains two segments $e_{i,j}, e_{k,l}$ such that $\{i, j, k, l\}$ are all different and $\overline{P_iP_j} \cap \overline{P_kP_l} = \{Q\}$.

Without loss of generality, let $c^* = P_i e_{i,j} P_j p_{j-k} P_k e_{k,l} P_l p_{l-i} P_i$. Consider now the tour $c^\dagger = P_i e_{i,k} P_k p_{k-j} P_j e_{j,l} P_l p_{l-i} P_i$, where p_{k-j} is the path p_{j-k} reversed.

The difference $L(c^*) - L(c^\dagger) = w(e_{i,j}) + w(e_{k,l}) - w(e_{i,k}) - w(e_{j,l}) = (d(P_i, Q) + d(Q, P_j)) + (d(P_k, Q) + d(Q, P_l)) - d(P_i, P_k) - d(P_j, P_l) \geq 0$ applying the triangle inequality to the triangles (P_i, P_k, Q) and (P_j, P_l, Q).

Considering that the points (P_i, P_k, Q) and (P_j, P_l, Q) are not aligned, the inequality becomes strict, so $L(c^\dagger) < L(c^*)$, that contradicts the fact that c^* is optimal. $\square$

3.3.1 Introducing nocrossing constraint

From Theorem 3.3.1 follows that during the search for an optimal TSP it is possible to avoid those Hamiltonian paths that include crossing edges (properties like this are usually referenced as *dominance rule* [113]). For this reason, the first constraint that is proposed is the `nocrossing` constraint, which imposes that a pair of segments in the TSP should not cross each other. In the successor representation, it is defined as follows:

$$\begin{aligned} &\texttt{nocrossing}(i, Next_i, j, Next_j) = \\ &= \left\{(n_i, n_j) \in \mathcal{D}(Next_i) \times \mathcal{D}(Next_j) | \left(\overline{P_i P_{n_i}} \cap \overline{P_j P_{n_j}}\right) \subset \{P_i, P_j\}\right\}^1. \end{aligned} \tag{3.4}$$

The `nocrossing` constraint presented in Equation 3.4 assures that segments $\overline{P_i P_{Next_i}}$ and $\overline{P_j P_{Next_j}}$ do not cross each other.

The `nocrossing` constraint is a *binary* constraint, i.e., it involves exactly two variables, since i and j are ground values when the constraint is imposed. It is also a *redundant* (or *implied*) constraint as its inclusion in the model does not change the set of solutions. Assuming that the problem being considered is a graph with n nodes, we would normally need to introduce $(n(n-1))/2$ constraints, one for each pair of nodes in the graph. The constraint can be implemented thorough a pair of propagators: one propagator removes values from the domain of $Next_i$ based on the values in the domain of $Next_j$, while the other propagator propagates changes on the other way.

The `nocrossing` constraint propagator could be implemented naively, for example using the table constraint [198, 36, 84, 128, 116] or the `propia` library [127]. A table constraint consists of a table (usually a list of tuples) of values that the involved variables must, or must not, assume. However, these implementations would be inefficient, due to the fact the constraint wakes up most of the time without being able to propagate. In addition, with the table constraint one should initially compute large tables, containing, for all pairs of edges in the graph, if they cross or not.

A naive propagator also tends to wake up every time a value is removed from the domain of $Next_i$ and would remove inconsistent values from the domain of $Next_j$. Propagating such constraint, with a naive propagator, would have the usual cost of arc consistency propagation for a single constraint of $O(d^2)$ (if d is the size of the domains) in each activation of the constraint.

From the definition of arc consistency and Equation 3.4, a value v can be removed from $\mathcal{D}(Next_j)$ only if $\overline{P_j P_v}$ intersects all possible segments originating

[1]We assume that the successor representation includes the `alldifferent`$(Next)$ constraint, so the variables $Next_i$ and $Next_j$ cannot assume the same value. The only type of intersection allowed at this point is on one of the extremes i.e. $Next_i = j$ or $Next_j = i$.

from P_i. A necessary condition for this is that all segments originating from P_i lie on the same half-plane with respect to the line $\overleftrightarrow{P_iP_j}$.

Theorem 3.3.2. *Let $o, u \in \mathcal{D}(Next_i)$ such that P_o and P_u do not lie on the line l passing through P_i and P_j and are in different half-planes with respect to the line l. Then, any value $k \in \mathcal{D}(Next_j)$ such that $P_k \notin l$ is arc consistent with respect to the constraint* `nocrossing`$(i, Next_i, j, k)$.

Proof. By contradiction, suppose there exists $k \in \mathcal{D}(Next_j)$ that is inconsistent with the constraint `nocrossing`$(i, Next_i, j, k)$. Since it is inconsistent, the segment $\overline{P_jP_k}$ must cross all the segments $\overline{P_iP_z}$ such that $z \in \mathcal{D}(Next_i)$, so in particular it crosses both $\overline{P_iP_o}$ and $\overline{P_iP_u}$. But the intersection $I_o \equiv \overline{P_iP_o} \cap \overline{P_jP_k}$ lies on a different half-plane from the one that hosts $I_u \equiv \overline{P_iP_u} \cap \overline{P_jP_k}$, so P_k lies in both half-planes, meaning that $P_k \in l$: absurd. □

Theorem 3.3.2 does not cover the case in which one of the points P_o, P_u lies on the line $\overleftrightarrow{P_iP_j}$. The following proposition deals with the cases where three points are aligned.

Proposition 3.3.3. *Given a graph G whose nodes do not lie all on the same line; let a, b and c be three nodes of G such that $P_c \in \overline{P_aP_b}$. Then, segment $\overline{P_aP_b}$ is not in the optimal TSP.*

Proof. The proof is similar to that of Theorem 3.3.1 as this proposition is no more than a special case. Let P_c be the point that lies on segment $\overline{P_aP_b}$, by contradiction suppose that segment $\overline{P_aP_b}$ appears in the optimal TSP.

The optimal TSP c^* contains two points: one immediately preceding P_c which we will denote by $P_{c'}$ and one immediately following which we will denote by $P_{c''}$. Let $c^* = P_a e_{a,b} P_b p_{b-c'} P_{c'} e_{c',c} P_c e_{c,c''} P_{c''} p_{c''-a}$.

Now, point P_c must be on segment $\overline{P_{c'}P_{c''}}$, if the points were not aligned it would be better to take segments $\overline{P_{c'}P_{c''}}$, $\overline{P_aP_c}$ and $\overline{P_cP_b}$.

Consider now the tour $c^\dagger = P_a e_{a,c'} P_{c'} p_{c'-b} P_b e_{b,c} P_c e_{c,c''} P_{c''} p_{c''-a}$. The difference $c^* - c^\dagger = w(e_{a,b}) + w(e_{c',c}) - w(e_{a,c'}) - w(e_{b,c})$. At the beginning, we assumed that P_c is a point on segment $\overline{P_aP_b}$, so we can rewrite $w(e_{a,b})$ as $w(e_{a,c}) + w(e_{c,b})$ obtaining $c^* - c^\dagger = w(e_{a,c}) + w(e_{c',c}) - w(e_{a,c'}) = d(P_a, P_c) + d(P_{c'}, P_c) - d(P_{c'}, P_c)$ but for triangular inequality this quantity is positive, contradicting the fact that segment $\overline{P_aP_b}$ can be part of c^*.

□

Given Proposition 3.3.3, it is possible to remove from the domain of each variable $Next_a$ all the values b such that the segment $\overline{P_aP_b}$ contains another node of the graph G. For the remainder of this thesis, we will assume that this pre-processing step has been performed before the search begins; this assumption simplifies the following discussion.

Algorithm 4 `nocrossing` propagator

```
1: function NOCROSSING_PROPAGATOR($i, Next_i, j, Next_j$)
2:     $Under \leftarrow$ select one element in $\mathcal{D}(Next_i)$ s.t. $P_{Under}$ is under the line $\overleftrightarrow{P_iP_j}$
3:     if there is no such element then
4:         NO_CROSSING_PROPAGATOR_PHASE_2($i, Next_i, j, Next_j$)
5:     else $Over \leftarrow$ select one element in $\mathcal{D}(Next_i)$ s.t. $P_{Over}$ is over the line $\overleftrightarrow{P_iP_j}$
6:         if there is no such element then
7:             NO_CROSSING_PROPAGATOR_PHASE_2($i, Next_i, j, Next_j$)
8:         else suspend waiting for either $Over$ or $Under$ to be removed from $\mathcal{D}(Next_i)$
9: function NOCROSSING_PROPAGATOR_PHASE_2($i, Next_i, j, Next_j$)
10:    $\underline{\alpha} \leftarrow \min\{\alpha_x | x \in \mathcal{D}(Next_i)\}$
11:    Let $x^i_{\underline{\alpha}}$ the value in $\mathcal{D}(Next_i)$ corresponding to $\underline{\alpha}$
12:    $\overline{\beta} \leftarrow \max\{\beta_x | x \in \mathcal{D}(Next_i)\}$
13:    Let $x^i_{\overline{\beta}}$ the value in $\mathcal{D}(Next_i)$ corresponding to $\overline{\beta}$
14:    for all $y^j \in \mathcal{D}(Next_j)$ s.t. $\alpha_{y^j} < \underline{\alpha}$ do
15:        if $\beta_{y^j} > \overline{\beta}$ then
16:            remove $y^j$ from $\mathcal{D}(Next_j)$
17:    if $|\mathcal{D}(Next_i)| > 1 \ \wedge \ |\mathcal{D}(Next_j)| > 1$ then
18:        suspend waiting for either $x^i_{\underline{\alpha}}$ or $x^i_{\overline{\beta}}$ to be removed from $\mathcal{D}(Next_i)$
```

Algorithm 4 sketches the algorithm of a propagator for the `nocrossing` constraint; it is awakened when the domain of variable $Next_i$ is reduced, and it performs propagation to possibly reduce the domain of $Next_j$; to fully implement the constraint, another symmetric propagator would be imposed on the reverse direction (from $Next_j$ to $Next_i$).

As long as the value j is contained in the domain of the variable $Next_i$, according to Theorem 3.3.1, no propagation can take place because the point P_j lies on the same straight line as the point P_i ; for this reason, the propagator for the variable $Next_i$ is added to the constraint store when the value j is removed from $\mathcal{D}(Next_i)$.

When added to the constraint store the propagator is suspended and waits for all elements in the domain of $Next_j$ to lie in the same half-plane with respect to $\overleftrightarrow{P_iP_j}$. To do so, one element $Over \in \mathcal{D}(Next_j)$ and one $Under \in \mathcal{D}(Next_j)$ that lie, respectively, over and under the line $\overleftrightarrow{P_iP_j}$ are selected. If one of them does not exist, all possible segments originating from P_j lie on the same half-plane and the control passes to the next phase; otherwise, the propagator suspends for one of the two $Over$ and $Under$ elements to be removed from $\mathcal{D}(Next_j)$. This strategy mimics, in a sense, the idea of watched literals proposed in SAT solvers [151] (and currently used also in other solvers).

Checking if a point P_a lies in the half-plane under or over the line $\overleftrightarrow{P_bP_c}$ amounts to check the sign of the (projection on the z axis) of the cross product $(P_a - P_b) \times (P_c - P_b)$.

Exploiting the approach just illustrated, which relies on Theorem 3.3.2, permits to significantly reduce the number of activations of the propagator.

When all segments in the domain of the variable $Next_i$ lie on the same half-plane with respect to the line $\overleftrightarrow{P_iP_j}$ we should, for each element $x \in \mathcal{D}(Next_j)$, check if the segment $\overline{P_jP_x}$ crosses all segments $\overline{P_iP_y}$ where $y \in \mathcal{D}(Next_i)$ i.e., all segments exiting from the point P_i. Naively performing the crossing check would require a number of operations equal to $O(n^2)$ each time the propagator wakes up (typically an exponential number of times while exploring the search space).

In order to reduce the number of crossing check, it is possible to sort the elements $P_x \in \mathcal{D}(Next_i)$ from smallest to largest according to the $\angle P_iP_jP_x$ angle. Since the points in the domain of the variable $Next_i$ are ordered according to angles, we have to check whether there is a point P_t in the domain of the variable $Next_j$ so that the angle $\angle P_iP_jP_t$ is smaller than the angle $\underline{\alpha} = \min\{\alpha_x | x \in \mathcal{D}(Next_i))\}$: if such a point P_t exists, then the segment P_jP_t could intersect all segments originating from P_i so we perform the crossing test; if it does not exist then there is no value to be removed from the $Next_j$ domain and we can suspend the propagator waiting for the value corresponding to $\underline{\alpha}$ to be removed.

This turns out to be a necessary condition for the arc consistency of the `nocrossing` propagator and is further detailed in Theorem 3.3.4.

Theorem 3.3.4. *For each $k \in \mathcal{D}(Next_i) \cup \mathcal{D}(Next_j)$, let $\alpha_k = \angle P_iP_jP_k$. If all the elements $q \in \mathcal{D}(Next_i)$ lie on the same half-plane with respect to the line $\overleftrightarrow{P_iP_j}$, then a necessary condition for a segment $\overline{P_jP_t}$ originating from P_j and reaching an element $t \in \mathcal{D}(Next_j)$ to cross all segments $\overline{P_iP_q}$ originating from P_i is that $\alpha_t \leq \underline{\alpha}$, where $\underline{\alpha} = \min\{\alpha_q \mid q \in \mathcal{D}(Next_i)\}$.*

Proof. Consider a coordinate system centered into P_j, with the abscissa pointing toward P_i and such that all the points in $\mathcal{D}(Next_i)$ have non-negative ordinate (see Figure 3.2).

By contradiction, suppose that $\alpha_t > \underline{\alpha}$; we prove that there is a segment originating from P_i that does not intersect with $\overline{P_jP_t}$. Let $z \in \mathcal{D}(Next_i)$ such that $\underline{\alpha} = \angle P_iP_jP_z$.

In polar coordinates, the segment $\overline{P_iP_z}$ is seen from P_j with angles between 0 and $\underline{\alpha}$. All the points on the segment $\overline{P_jP_t}$ are seen under the angle α_t. Since $\alpha_t > \underline{\alpha}$, there is no intersection between $\overline{P_iP_z}$ and $\overline{P_jP_t}$. □

A vector representation was used to calculate the angles. Suppose we want to compute the angle $\angle P_iP_jP_k$. Given the coordinates of points $P_i = (x_i, y_i)$, $P_j = (x_j, y_j)$ and $P_k = (x_k, y_k)$, it is possible to determine the vectors $\boldsymbol{u} = (u_1, u_2) = (x_j - x_i, y_j - y_i)$, $\boldsymbol{v} = (v_1, v_2) = (x_k - x_i, y_k - y_i)$. Note that both vectors have their tails at point P_i as it must coincide with the vertex of the angle we wish to calculate. Calculating the angle $\angle P_iP_jP_k$ now corresponds to

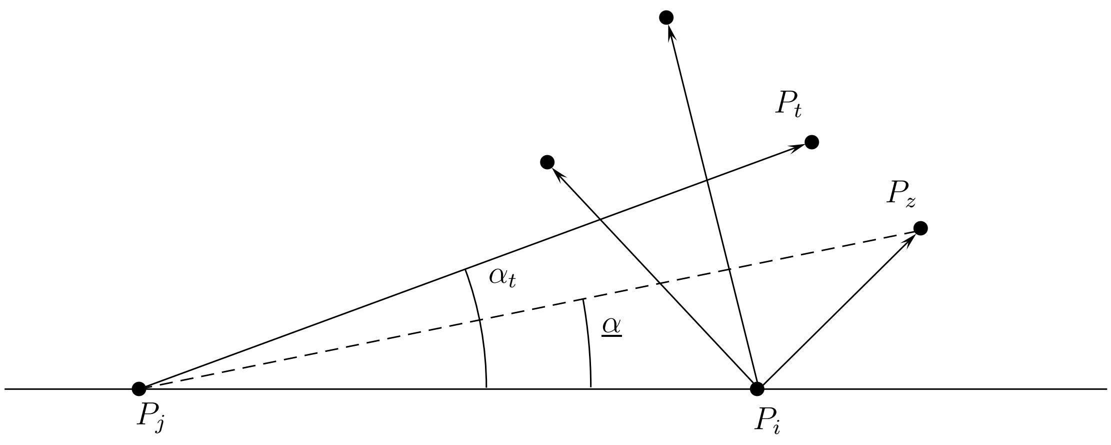


Figure 3.2: An arrow from P_x to P_y means that $y \in \mathcal{D}(Next_x)$. Dashed lines are plotted to show the angles.

calculating the angle between the two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$. In our implementation we used the following Equation 3.5.

$$\angle P_i P_j P_k = atan2\left(\boldsymbol{u} \times \boldsymbol{v}, \boldsymbol{u} \cdot \boldsymbol{v}\right). \tag{3.5}$$

where $\boldsymbol{u} \cdot \boldsymbol{v}$ is the dot product (see Eq. 3.6) and $\boldsymbol{u} \times \boldsymbol{v}$ is the cross product (see Eq. 3.7).

$$\boldsymbol{u} \cdot \boldsymbol{v} = u_1 v_1 + u_2 v_2 \tag{3.6}$$

$$\boldsymbol{u} \times \boldsymbol{v} = u_1 v_2 - u_2 v_1 \tag{3.7}$$

The magnitude of the angle $\angle P_i P_j P_k$ is expressed in radians in the interval $(-\pi, \pi)$. We note that the function $atan2$ is a variation of the arctangent that can be defined for all pairs of real values except the pair $(0, 0)$ and is defined in terms of the classical arctangent as given in Equation 3.8.

$$atan2(y,x) = \begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0, \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0, \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0, \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0, \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0, \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases} \tag{3.8}$$

Since the condition of Theorem 3.3.4 is only necessary, if it is verified for a particular point P_k it does not guarantee that the segment $P_j P_k$ actually crosses all segments that leave the node $Next_i$. A counterexample is shown in Figure 3.3. It is therefore still necessary to perform a crossing test, which can be implemented efficiently using the *Faster Line Segment Intersection* algorithm [12].

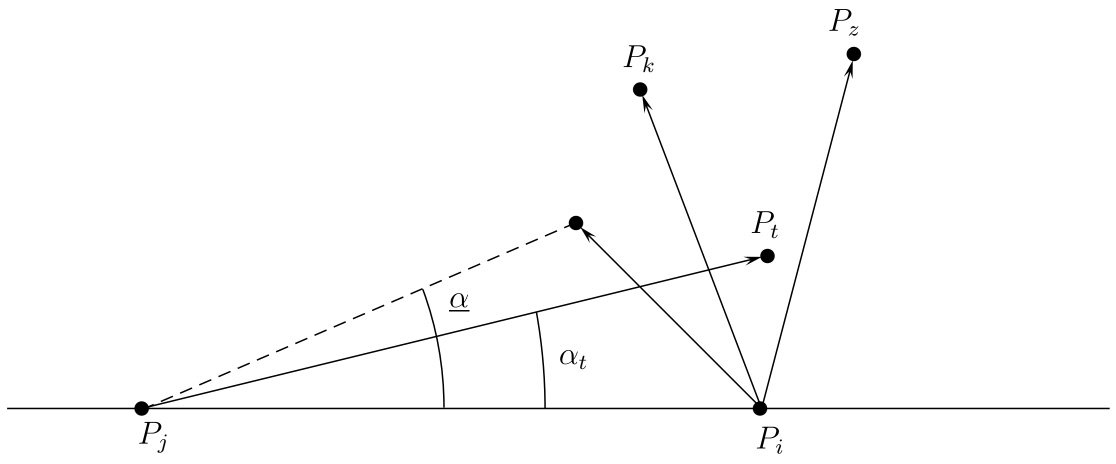


Figure 3.3: The condition in Thm 3.3.4 is not sufficient: $\alpha_t < \underline{\alpha}$ but $\overline{P_jP_t}$ does not cross all segments exiting from P_i.

By taking advantage of geometric information, especially angles, we were able to find an even more efficient method to check whether a crossing exists than using the Faster Line Segment Intersection algorithm. Theorem 3.3.5 is a sufficient condition for the existence of a crossing.

Theorem 3.3.5. *For each $k \in \mathcal{D}(Next_i) \cup \mathcal{D}(Next_j)$, let $\beta_k = \angle P_kP_iP_j$. Assume all the elements $q \in \mathcal{D}(Next_i)$ lie on the same half-plane with respect to the line $\overleftrightarrow{P_iP_j}$. Suppose there exists $t \in \mathcal{D}(Next_j)$ such that $\alpha_t < \underline{\alpha}$.*

Then a sufficient condition for segment $\overline{P_jP_t}$ to cross all segments $\overline{P_iP_q}$ such that $q \in \mathcal{D}(Next_i)$ is that $\beta_t > \overline{\beta}$, where $\overline{\beta} = \max\{\beta_q \mid q \in \mathcal{D}(Next_i)\}$.

Proof. By contradiction, suppose $\exists k \in \mathcal{D}(Next_i)$ such that $\overline{P_iP_k}$ does not intersect $\overline{P_jP_t}$ (Figure 3.4). Since $\alpha_t < \underline{\alpha} \leq \alpha_k$, P_k and P_i lie on different sides of the ray $\overrightarrow{P_jP_t}$.

In order not to have a crossing between $\overline{P_iP_k}$ and $\overline{P_jP_t}$, both P_j and P_t must lie on the same half-plane with respect to $\overrightarrow{P_iP_k}$. Since $\beta_j = 0$, and all points in the domain of $Next_i$ are above the x axis, $\beta_j < \beta_k$, thus to be on the same half-plane, also $\beta_t < \beta_k$. But $\beta_t > \overline{\beta} \geq \beta_k$: contradiction. □

Thanks to Theorems 3.3.4 and 3.3.5 we can get a better complexity compared to the naive algorithm. Note that all angles can be pre-computed before starting the search, which avoids the computation of trigonometric functions during the search. It is also possible to pre-compute the elements in the (initial) domains of the two variables, sorted according to their angle α, and then store them in a linked list. In this way, the computation of the minimum in line 10 of Algorithm 4 is equivalent to finding the first element in the linked list that belongs to $\mathcal{D}(Next_i)$; Assuming that the domain membership is checked in constant time, the minimum on line 10 can be found in $O(d)$ amortized time on one branch of the search tree. Get a linked list with the elements in $\mathcal{D}(Next_i)$ ordered by their angle β: line 12 is also executed

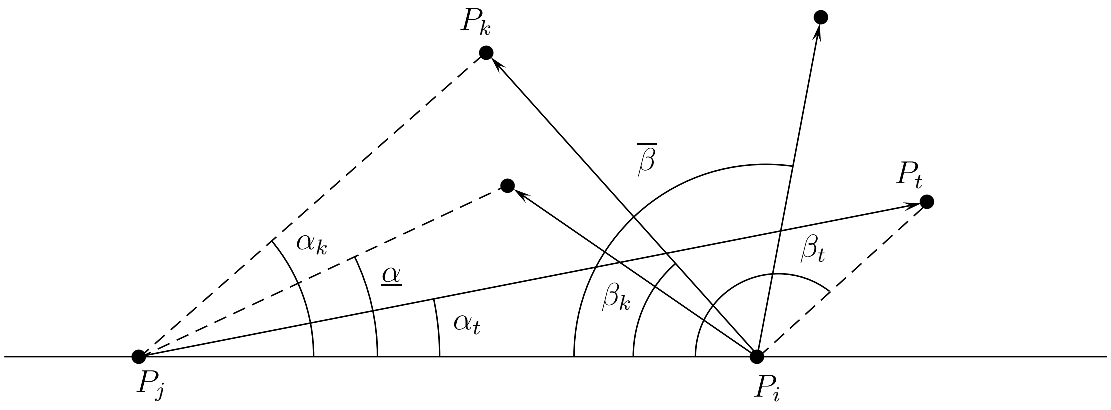


Figure 3.4: An arrow from P_x to P_y means that $y \in \mathcal{D}(Next_x)$. Dashed lines are plotted to show the angles.

in $O(d)$ amortized time on a branch of the search tree. The loop in lines 14 - 16 searches the linked list and stops as soon as an element is found with an angle greater than or equal to $\underline{\alpha}$; assuming that the removal of a domain element takes place in constant time, and since the comparison on line 15 takes a constant time, the entire loop has $O(d)$ complexity for each activation of the propagator of phase 2 (to be compared with the $O(d^2)$ of the naive propagator). Since the propagator is activated when at least one element from $\mathcal{D}(Next_i)$ is removed, this propagator is woken up at most $O(d)$ times in a branch of the search tree, which gives phase 2 a general complexity of $O(d^2)$ amortized time in one branch of the search tree.

3.3.2 Convex hull reasoning and clockwise constraint

A useful consequence of Theorem 3.3.1 is given in the following Corollary 3.3.6 and is based on the concept of convex hull. The *convex hull* of a set of points in a Euclidean space is the minimum convex set containing all the points. In the plane it corresponds to a convex polygon, and it is completely defined by its vertices.

Corollary 3.3.6 (Deineko, van Dal, and Rote 1994 [59])**.** *"Assuming that not all cities lie on one line, an optimal tour has the property that the cities on the boundary of the convex hull of the cities are visited in their cyclic order".*

Given a set of points P, we denote with $\mathcal{H}(P) = \langle H_0, H_1, \ldots, H_{|\mathcal{H}(P)|-1} \rangle$ the sequence of vertexes on the boundary of the convex hull in clockwise order. To compute the convex hull $\mathcal{H}(P)$, we used the widely known Andrew's monotone chain algorithm [10], with complexity $O(n \log n)$. The algorithm calculates the upper and lower hull with a complexity of $O(n)$ which are then combined to form the convex hull. However, this requires that the points are first sorted with respect to their x-coordinates, and also with respect to their y-coordinates in case of a tie, hence the complexity already mentioned.

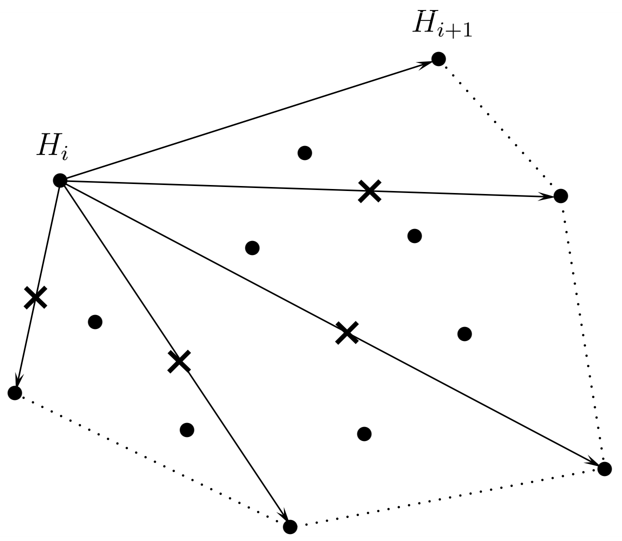


Figure 3.5: The successor of a convex hull vertex cannot be another vertex on the boundary of the convex hull except for the one that immediately follows it.

To simplify the exposition, the following pruning is presented in the context of symmetry breaking constraints, although similar reasoning might be performed also while not breaking symmetries.

In the successor representation, the same TSP can be represented by two symmetrical solutions that differ just for the order (clockwise or counterclockwise) in which the nodes are visited. One way to break this symmetry is to fix one direction (e.g., clockwise); in such a case, the convex hull reasoning is an efficient way to impose the clockwise order.

We devised three ways to exploit the information about the hull for propagation.

The simplest is to impose that the successor of a convex hull vertex cannot be another vertex member of $\mathcal{H}(P)$ except for the one that immediately follows it, see Equation 3.9.

$$\forall i \in [0, |\mathcal{H}(P)| - 1],\ \mathcal{D}(Next_{H_i}) \cap \mathcal{H}(P) \subseteq \{H_{(i+1) \mod |\mathcal{H}(P)|}\} \tag{3.9}$$

Equation 3.9 can be implemented as a unary constraint which in practice is equivalent to reducing the initial domain of the variables, so no overhead is produced during the search (see Figure 3.5).

The second way is reasoning on the angle formed by the incoming and the outgoing arcs in hull vertexes: in order to visit nodes in a clockwise order, the angle between the incoming edge and the outgoing edge of H_i cannot be positive (it must be between $-\pi$ and 0) or, stated otherwise, it must correspond to a right turn (see Figure 3.6).

The easiest implementation consists of waiting until $Next_h$ becomes ground; when the arc outgoing from P_h is fixed, the value h is removed from the domain

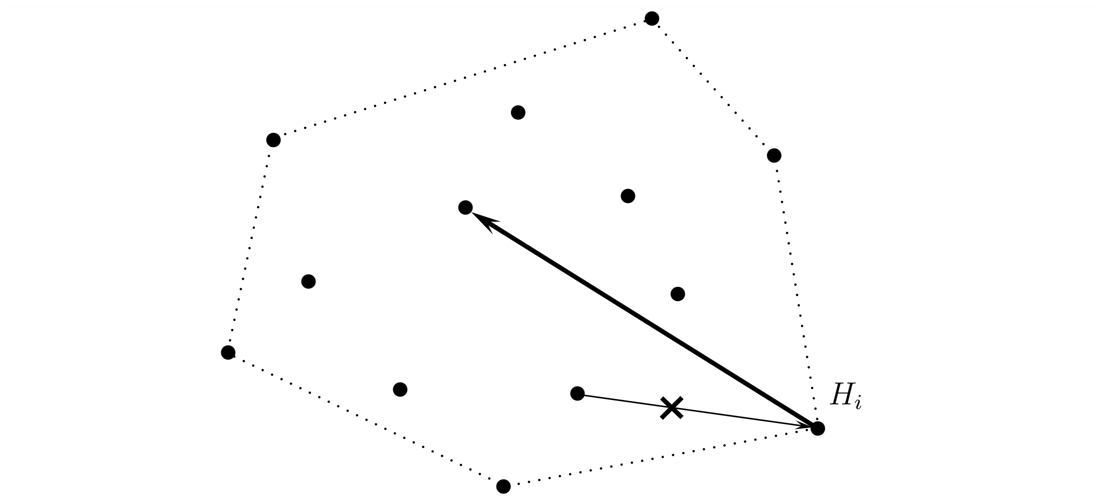


Figure 3.6: In order to visit nodes in a clockwise order, the angle between the incoming edge and the outgoing edge of a convex hull vertex cannot be positive (it must be between $-\pi$ and 0).

of all other variables $Next_i$ such that the angle $\angle P_i P_h P_{Next_h}$ would correspond to a left turn.

If the constraint model also contains the *Prev* variables, the angle formed by the outgoing edge and any reference direction must be smaller than the angle that the incoming edge forms with the same reference direction. This produces a propagator in the same spirit of the classical *less-than* propagator: simply compute the minimum angle in $\mathcal{D}(Next_h)$ and remove from $\mathcal{D}(Prev_h)$ the elements associated with a smaller (or equal) angle (Algorithm 5). A symmetric propagator takes care of the opposite direction (from $Prev_h$ to $Next_h$). Again, note that all angles are pre-computed before search, and the search for the minimum takes $O(d)$ amortized time over one branch of the search tree.

Algorithm 5 clockwise_angle_propagator

```
Require: $P_h$ to be on the boundary of the convex hull
1: function CLOCKWISE_ANGLE_PROPAGATOR($h$, $Next$, $Prev$)
2:     $m \leftarrow \min\{\alpha_v \mid v \in \mathcal{D}(Next_h)\}$
3:     $\mathcal{D}(Prev_h) \leftarrow \mathcal{D}(Prev_h) \setminus \{i \mid \alpha_i \leq m\}$
4:     if $Next_h$ and $Prev_h$ are not ground then
5:         suspend until $m$ is removed from $\mathcal{D}(Next_h)$
```

The third way results from stating that any path starting from a convex hull vertex cannot reach any other convex hull vertex except the one that directly follows it. Put it more precisely, each vertex in a path originating from a point H_i cannot reach any vertex in H except for $H_{(i+1) \bmod |\mathcal{H}(P)|}$ (see Figure 3.7). The propagator is imposed for each pair $(H_i, H_{(i+1) \bmod |\mathcal{H}(P)|})$. The implementation of this propagator is inspired by the `circuit` constraint [46] but performs more powerful pruning

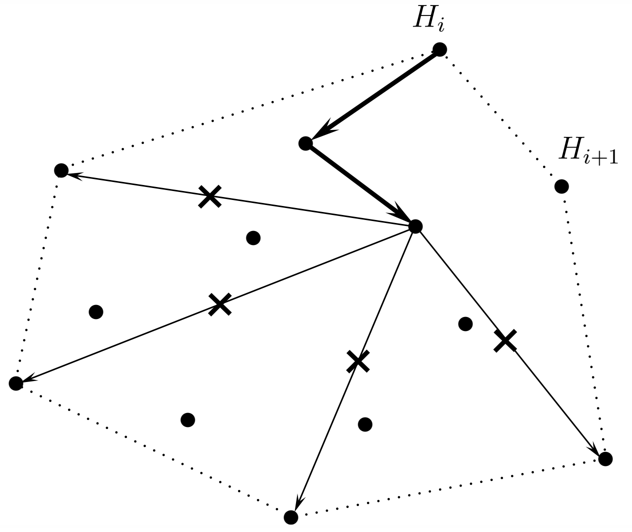


Figure 3.7: Each path originating from a convex hull vertex cannot reach any convex hull vertex except for the one immediately following it. Implementation is inspired by the `circuit` constraint [46] but performs more powerful pruning.

thanks to the convex hull reasoning. If a partial path has been defined starting from H_i up to a node j, and such path does not contain vertex $H_{(i+1) \bmod |\mathcal{H}(P)|}$, then the variable $Next_j$ cannot take any value in $\mathcal{H}(P)$ except for $H_{(i+1) \bmod |\mathcal{H}(P)|}$ (Algorithm 6). If the partial path reaches the next vertex of the convex hull, the constraint is entailed and exits the constraint store (line 3). In the propagator for the `circuit` constraint [46], instead, from the domain of the variable $Next_j$ only the initial value (in our case, H_i) of the path is removed.

Algorithm 6 hull_path_propagator

```
Require: $P_{End}$ to be on the boundary of the convex hull.
1: function HULL_PATH_PROPAGATOR($Start, End, \mathcal{H}(P), Next$)
2:     if $Start == End$ then
3:         return $true$
4:     if $Next_{Start}$ is ground then
5:         HULL_PATH_PROPAGATOR($Next_{Start}, End, \mathcal{H}(P), Next$)
6:     else
7:         remove $(\mathcal{H}(P) \setminus \{End\})$ from $\mathcal{D}(Next_{Start})$
8:     suspend waiting for $Next_{Start}$ to become ground
```

Extension of the convex hull reasoning

Now that we have propagators that exploit the knowledge about the convex hull, we wish to extend their applicability to the points within the hull as well.

One consequence of the absence of crossings is that the optimal TSP is a simple polygon, which is a closed polygonal chain of line segments that do not cross each other, and it divides the plane into exactly two areas: an *internal* and an *external* area.

Now imagine cutting the optimal TSP with two vertical lines (parallel to the y axis): the stripe between the two lines will contain alternate internal and external areas. The borders (i.e., the parts of the circuit inside the stripe) will be visited alternately from left to right (or clockwise) and from right to left (or counterclockwise). To exploit this informal intuition for pruning, we provide the following theorem:

Theorem 3.3.7. *Suppose that (e.g., during search) a partial path p_{s-e} has been defined, starting in node s and ending in node e. Consider the polygon Q delimited by such path and by the segment $\overline{P_sP_e}$, and suppose that such polygon is a simple polygon, i.e., no two edges intersect. Suppose that the partial path p_{s-e} touches its vertexes in clockwise order.*

Let $F \subset V$ be the set of nodes whose corresponding points lie in the interior of polygon Q, $I = F \cup \{s, e\}$ and let $\mathcal{H}(I) = \langle H_0^I \equiv e, H_1^I, \ldots, H_{k-1}^I, H_k^i \equiv s\rangle$ be the sequence of vertexes of its convex hull, in counterclockwise order.

Suppose that the convex hull $\mathcal{H}(I)$ does not intersect the path p_{s-e}, except for the endpoints P_s and P_e.

Then any non self-crossing tour containing the path p_{s-e} reaches the vertexes in $\mathcal{H}(I)$ in the order $H_0^I, \ldots H_k^I$.

Proof. By contradiction, suppose that a non self-crossing circuit $c^* \supseteq p_{s-e}$ reaches the vertexes of H^I in an order different from their sequence order. W.l.o.g., suppose that after vertex H_j^I, the next vertex of H^I reached by the tour c^* is H_{j+2}^I; i.e., vertex H_{j+1}^I is not reached in the order of H^I. Let $p_{H_j^I - H_{j+2}^I} \subset c^*$ be the path connecting H_j^I and H_{j+2}^I.

Consider the polygon R (dotted, in Fig. 3.8) delimited by:

(*i*) the path p_{s-e}
(*ii*) the perimeter of the convex hull $\mathcal{H}(I)$ from P_e to H_j^I
(*iii*) the path $p_{H_j^I - H_{j+2}^I}$
(*iv*) the perimeter of the convex hull $\mathcal{H}(I)$ from H_{j+2}^I to P_s.

Clearly, the point H_{j+1}^I lies in the interior of R. In order to reach it without self-crossings, c^* cannot pass through p_{s-e} nor $p_{H_j^I - H_{j+2}^I}$. On the other hand, c^* cannot cross the boundary of the $\mathcal{H}(I)$ between P_e and H_j^I (and from H_{j+2}^I to P_s) because, by definition of convex hull, there are no vertexes to be visited that are interior to polygon Q and that do not belong to $\mathcal{H}(I)$. So, c^* cannot reach H_{j+1}^I. □

If we find an *internal* hull $\mathcal{H}(I)$ (see Figure 3.8), we can then apply the previous propagators (Algorithms 5 and 6) also to the vertexes of $\mathcal{H}(I)$, with the obvious

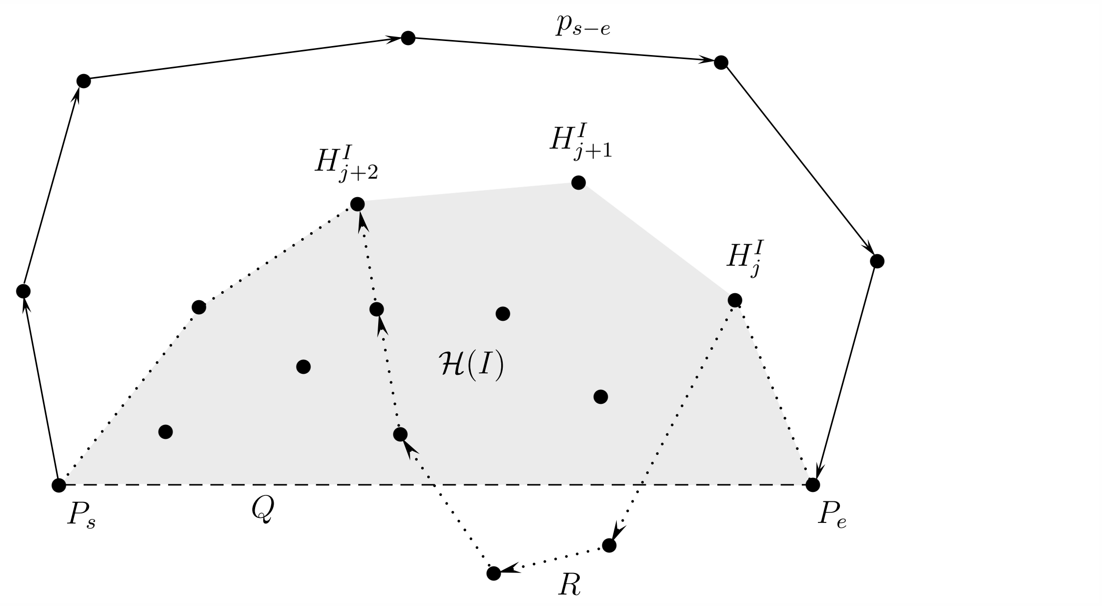


Figure 3.8: From Theorem 3.3.7: the path p_{s-e} is the current assignment. Polygon Q is delimited by p_{s-e} and the (dashed) segment $\overline{P_e P_s}$. I contains the points strictly inside Q plus P_s and P_e; $\mathcal{H}(I)$ (grey in the picture) is its convex hull. Polygon R is delimited by p_{s-e} and the dotted segments.

care that if p_{s-e} reaches points in clockwise order, then the vertexes of the $\mathcal{H}(I)$ will be reached in counterclockwise order (and vice versa). We maintain all partial paths during the search, and we compute a convex hull for each of these paths. In this way, we are orthogonal to heuristics (we can use any search heuristic without invalidating our propagation).

In the implementation, we applied the pruning on internal hulls only when the polygon Q is convex. Checking the convexity amounts to check that each turn in the path is on the same side (a right turn, on a clockwise path), it makes it easier to find when a point is inside the polygon and also allows us to avoid checking that $\mathcal{H}(I)$ does not intersect p_{s-e}.

The time complexity of our implementation of the extended convex hull is $O(n^2)$ to compute the points inside the polygon, then we use Andrew's monotone chain ($O(n \log n)$) [10] to find the hull. We currently recompute $\mathcal{H}(I)$ from scratch after each decision.

3.3.3 Experimental evaluation

To assess the effectiveness of the proposed algorithms, we devised experiments based on randomly generated TSPs and structured instances.

All algorithms are implemented in the ECL^{i}PS^e CLP language [184]. All constraint models are based on the successor representation described in the preliminaries (with the `circuit` constraint and `alldifferent` [165] for improved pruning) with both *Next* and *Prev* variables, that are linked through the `inverse` constraint.

The basic model, named `CLP(FD)` in the following, beside the `alldifferent`, `circuit` and `inverse` constraint, includes as symmetry breaking the constraint $Next_1 < Prev_1$ as in Benchimol *et al.* [27].

The constraint model named `GEOMETRIC` includes the `nocrossing` constraint, the removal of aligned points according to proposition 3.3.3, and the `clockwise` constraint that implements the propagation described in Section 3.3.2, which in this model also acts as a symmetry breaking constraint.

In order to show that the pruning we provide is not subsumed by that of state-of-the-art techniques, we implemented in ECL^{i}PS^e, in the successor representation, also the Held and Karp bound with pruning based on reduced and marginal costs, as proposed by Benchimol *et al.* [27] (shown with `BVHRRR` in the following).

As the focus was on pruning and not on search strategies, we use the *max-regret* [46] and the state-of-the-art `LC_FIRST MAX_COST` [69], based on *Last Conflict* [129]. As [27], we also experimented injecting the upper bound given by the Lin-Kernighan-Helsgaun (LKH) (v. 2.0.7) algorithm.

All tests were run on ECL^{i}PS^e v. 7.0, build #48, with a time limit of 1800s on Intel® Xeon® E5-2630 v3 CPUs running at 2.4GHz, using only one core and with 1GB of reserved memory.

Random Instances

To generate realistic instances, we used the generator of the DIMACS challenge [111], that provides instances in two classes: *uniform* and *clustered.* We randomly generated instances from 20 to 50 nodes in steps of 2, in both classes. For each size and class, we generated 30 instances. Uniform random generated instances consist of integer coordinate points uniformly distributed in a square of 10^6 side. Randomly generated instances consist of clusters of points, whose centers are uniformly distributed in a square of 10^6 side. Each point is then randomly associated with a cluster center and two normally distributed variables, each of which is then multiplied by $10^6/\#nodes$, rounded, and added to the corresponding integer coordinate of the chosen center to obtain the coordinates of the point.

Figure 3.9 shows the geometric mean of the runtime of the four algorithms varying the size of the instance and the search strategy. The addition of the filtering on geometric properties roughly halves the runtime, both with respect to the simple `CLP(FD)` and to the advanced pruning based on the Held and Karp (`BVHRRR`). Cactus plots (Figure 3.10) show that, when the LKH bound is used, the two search strategies perform in a quite similar way. In both graphs, the mark distinguishes the various search strategies, solid (opposed to dotted) lines represent usage (resp. not usage) of the `BVHRRR` pruning, while thick (resp. thin) lines represent usage (or not usage) of `GEOMETRIC` filtering. The introduction of

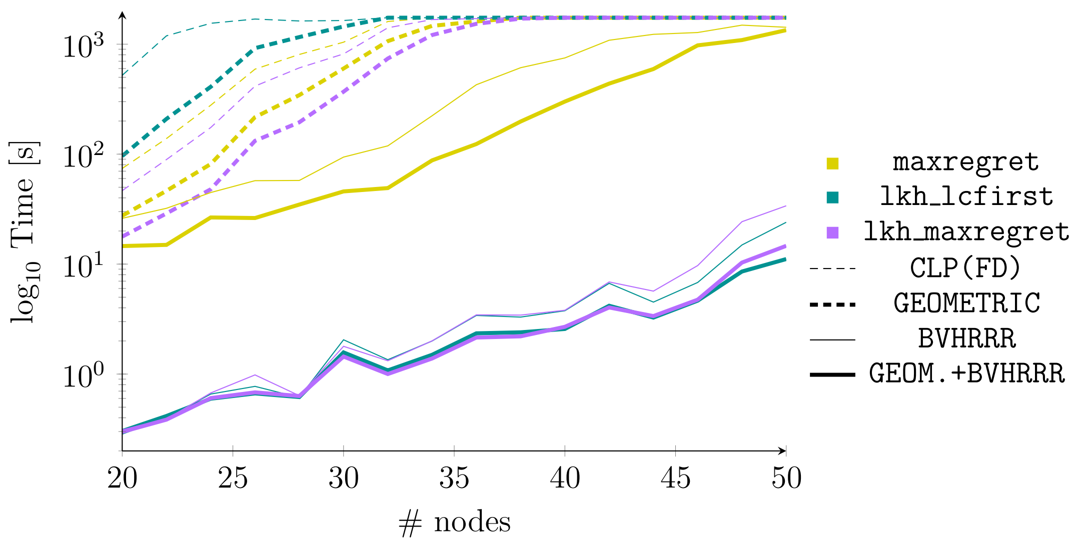


Figure 3.9: Experimental results on randomly-generated Euclidean TSP instances. Average solving time of filtering algorithms varying the size of the instances and the search strategies.

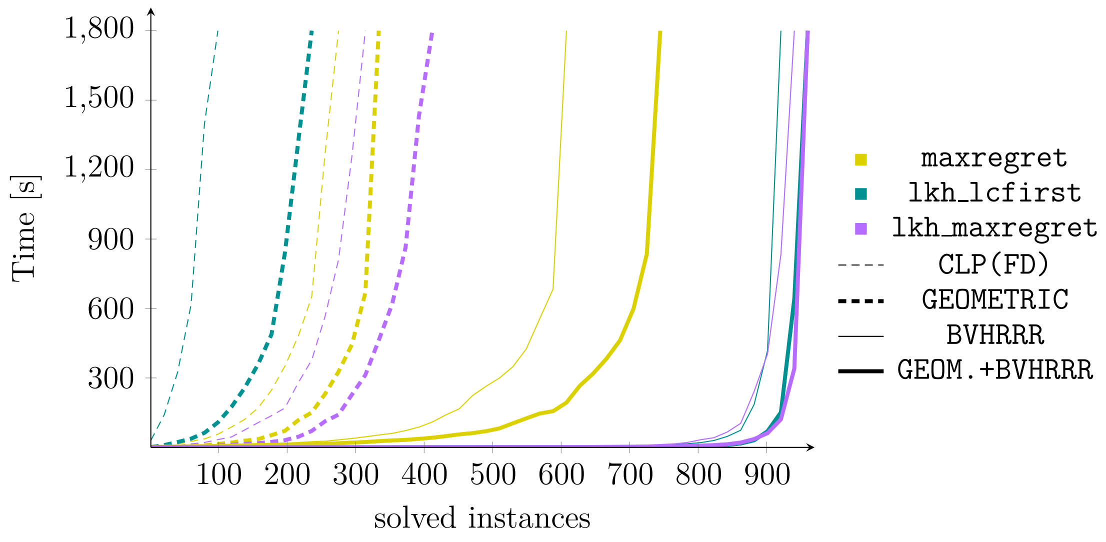


Figure 3.10: Experimental results on randomly-generated Euclidean TSP instances. Number of solved instances varying the solving time.

geometric filtering (represented with thick lines) allowed us to solve almost all the instances within the given timeout.

Structured Instances

We considered all the instances taken from the TSPLIB, the Concorde website[2] and the CITIES dataset[3] up to 100 nodes. These sources provide various types of

[2]`http://www.math.uwaterloo.ca/tsp/world/countries.html`
[3]`https://people.sc.fsu.edu/~jburkardt/datasets/cities/cities.html`

instances, including Euclidean, represented as sets of points on the plane, and geographic, represented as sets of points (with latitude and longitude) on the surface of the earth. We selected all the Euclidean instances and also added those geographical instances in which the cities to be visited lie on a limited part of the geoid, so that the geographical distance can be approximated with the Euclidean distance.

Table 3.1 reports the results; we omit the instances where no algorithm could reach the optimal solution within the 1800 seconds time limit. Aside from simple instances, the constraint models that contain geometric filtering are the ones that optimally solve the instances in the shortest time. These results show the positive interaction between the geometric filtering and that carried out by `BVHRRR` alone. By analysing the instances that were not solved to optimality, we found that our additional pruning is more effective during the proof of optimality, rather than on finding good solutions.

Table 3.1: Comparing filtering algorithms on structured instances with time limit 1800s. For each instance we report total solving time and number of explored nodes to reach the optimal solution and prove its optimality.

	lkh_lcfirst				lkh_maxregret			
	BVHRRR		Geo+BvH		BVHRRR		Geo+BvH	
instance	time	nodes	time	nodes	time	nodes	time	nodes
uk12	**0.04**	12	0.05	12	**0.04**	**0**	**0.04**	**0**
burma14	0.06	14	0.07	14	**0.05**	**0**	0.07	**0**
ulysses16	**0.07**	16	0.09	16	**0.07**	**0**	0.08	**0**
ulysses22	**0.12**	22	0.15	22	**0.12**	**0**	0.16	**0**
wg22	0.14	22	0.18	22	**0.13**	**0**	0.17	**0**
bayg29	0.45	30	0.44	35	**0.41**	3	0.43	**2**
wi29	**0.23**	29	0.32	29	**0.23**	**0**	0.32	**0**
dj38	**0.42**	38	0.65	38	**0.42**	**0**	0.65	**0**
dantzig42	2.16	48	**1.60**	43	4.83	28	2.11	**3**
att48	3.21	66	**2.38**	55	8.02	36	3.10	**10**
uscap50	**0.64**	50	1.22	50	**0.64**	**0**	1.22	**0**
eil51	T/O	-	523.92	1384	313.53	916	**84.43**	**195**
berlin52	0.86	52	1.67	52	**0.84**	**0**	1.52	**0**
kn57	8.53	180	5.67	120	8.24	38	**5.14**	**18**
wg59	**1.13**	59	1.92	59	1.35	1	1.80	**0**
st70	1411.63	3934	**327.79**	**985**	T/O	-	T/O	-
eil76	72.56	227	33.20	156	160.09	230	**31.80**	**26**
rat99	T/O	-	T/O	-	T/O	-	**436.97**	**574**
rd100	25.81	117	**21.79**	107	52.99	80	21.94	**22**

3.4 Prediction of effective nocrossing constraints

In Section 3.3.1, to speed up the solving of Euclidean TSPs, we introduced the `nocrossing` constraint that avoids, during search, solutions that include crossing edges. As mentioned earlier, this constraint should be imposed for each pair of nodes, which results in a quadratic number of constraints. Note that these constraints are only intended to make the solution more efficient, they are not required for its correctness; in fact, they are redundant constraints, and it is well known in the CP literature that redundant constraints can improve inference efficiency [77]. One question might be whether all of these constraints do an effective pruning that reduces search space, or whether only some of them are really useful while others do not perform significant pruning while introducing overhead. In brief, to answer this question, we can say that in the experiments we tend to determine that not all the `nocrossing` constraints are equally useful: by experimental analysis a number of them offer a speed-up, whereas others solely introduce overhead. In the following, we will discuss how we evaluated the performance of every constraint and afterward the machine learning techniques we introduced to limit the overhead and maintain high performance.

3.4.1 Combining Constraint Programming with Machine Learning

There is a wide literature on approaches in which constraint programming is combined with machine learning and data mining [34].

One of the main ideas is portfolio selection (see, e.g., the survey [122] and references therein): given a set of algorithms (or solvers) that solve a same problem, select the best one for solving a given instance. The approach is based on obtaining data about the running time of the algorithms on a high number of instances.

Also, for each instance a number of features are computed, hopefully synthesizing those characteristics that make it easy or hard to solve.

After that, a classifier is learned trying to predict, given the set of features of a new unseen instance, which of the available solvers will be the fastest for that specific instance. Once a new instance is provided, the classifier chooses the best solver.

Although less strictly related to this thesis, we cite other approaches to combine machine learning and CP, including Empirical Model Learning [138, 137], trying to learn some features of a physical system and including its input/output relation as a new constraint, or approaches that try to learn single constraints or a whole constraint model given examples from the user [37, 23]. There have also been approaches where machine learning has been used to automate constraint design decisions, based on the problem to be solved, in order to increase the performance of constraint solvers [85].

3.4.2 Performance evaluation of nocrossing constraints

To evaluate the performance of each constraint we collected data while solving Euclidean TSP instances. In order to have a statistically significant number of instances, we used randomly-generated ones from multiple TSP generators.

We generate instances from 3 different classes. The first class, named *uniform*, is that of the random uniform Euclidean instances obtained by placing points uniformly distributed in a $10^6 \times 10^6$ square. We used the generator of the DIMACS challenge [111]. The second set of instances consists of *clustered* instances. Clustered instances have been generated through the R-package `netgen` [41] and its function `generateClusteredNetwork`. The function `generateClusteredNetwork` initially generates cluster centers by Latin Hypercube Sampling (LHS) to ensure that the clusters are placed separately from each other. Then it distributes points to the clusters according to a normal distribution that exploit the centers of the clusters as the mean vector and the distance to the center of the nearest neighbour cluster as the variance. The last type of generated instances is a combination of the two above. Morphed instances are in fact obtained by combining two TSP instances with the same number of nodes. Each *morphed* instance is generated by applying a convex combination to the coordinates of node pairs. The morphed instances were also obtained using the R-package `netgen` and in particular the function `morphInstances`.

For each `nocrossing` constraint (propagator), in each instance, we measured 3 indicators:

- the number of times the constraint has been resumed ($N_{activations}$);
- the number of values removed from the domains of the variables involved in the constraint as result of domain filtering (N_{pruned});
- the number of failures (and therefore backtracks) resulting from domain filtering.

The first two indicators were then combined to obtain a fourth one. This fourth indicator denoted as `RTIO` was calculated as the ratio $N_{pruned}/N_{activations}$. A constraint with a low `RTIO` wakes up many times without being able to prune values from the domains of the involved variables, which leads to an undesirable computational overhead, while a constraint with a high `RTIO` can perform a much stronger pruning compared to the number of activations and therefore it is worth imposing it.

Figures 3.11, 3.12 and 3.13 graphically show the number of value deletions, the number of failures and the `RTIO` respectively, for a typical instance of Euclidean TSP. In each figure, the darker the colour of the line, the higher the value of the corresponding indicator. We created many of these figures in the hope of finding a meaningful pattern that could help us identify useful or useless cases of `nocrossing` constraints. Unfortunately, we could not observe any interesting pattern, so we decided to introduce a machine learning step, which is described in more detail in the next section.

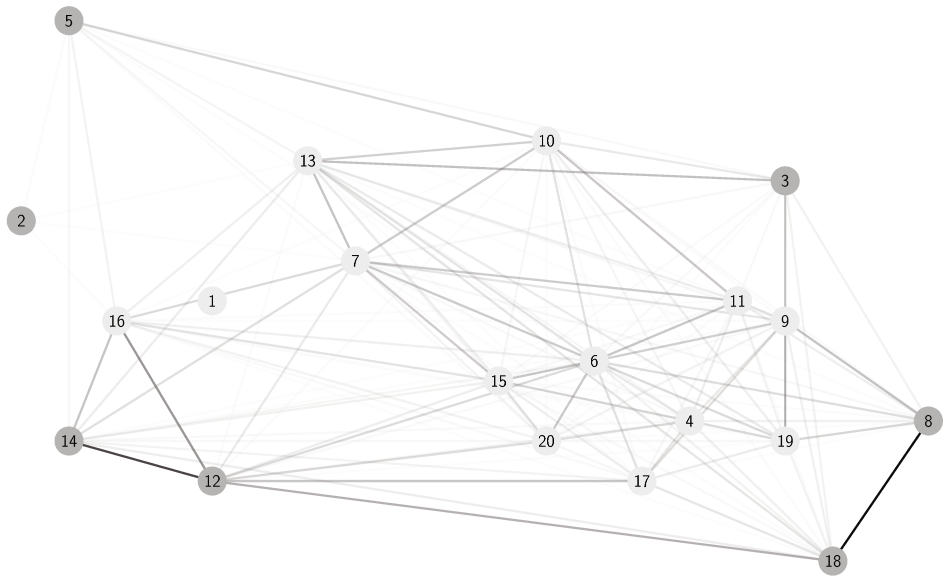


Figure 3.11: Graphical representation of the number of value deletions performed by each `nocrossing` constraint in a Euclidean TSP instance. The darker the colour of the segment, the higher the number of value deletions. Darker nodes are those on the boundary of the convex hull.

While in a preliminary version of this research [25] we arbitrarily choose to consider as useful all constraints in an instance $\mathcal{I}$ having `RTIO` greater than the average for their instance $\mu_{\mathcal{I}}^{\texttt{RTIO}}$, afterwards we decided to adopt a threshold $\theta^{\texttt{RTIO}}$, and postpone to the experimental evaluation the decision of its most suitable value [24]. More precisely, given a threshold $\theta^{\texttt{RTIO}}$, a $\texttt{nocrossing}(i, Next_i, j, Next_j)$ constraint is labeled as *useful* in an instance $\mathcal{I}$ if

$$\texttt{RTIO}_{\mathcal{I}}^{i,j} \geq \theta^{\texttt{RTIO}} \cdot \mu_{\mathcal{I}}^{\texttt{RTIO}}$$

and *useless* otherwise[4].

The relation we wish to learn could be seen as a function mapping each pair of points (in a generic Euclidean TSP instance) to the set $\{useful, useless\}$. In principle, each point could only be identified by its coordinates, but this could be a too specific information: the effectiveness of a constraint should be independent of

[4]This way of proceeding is also an arbitrary decision and in any case an approximation as there is no generally accepted way of assessing the performance of a propagator.

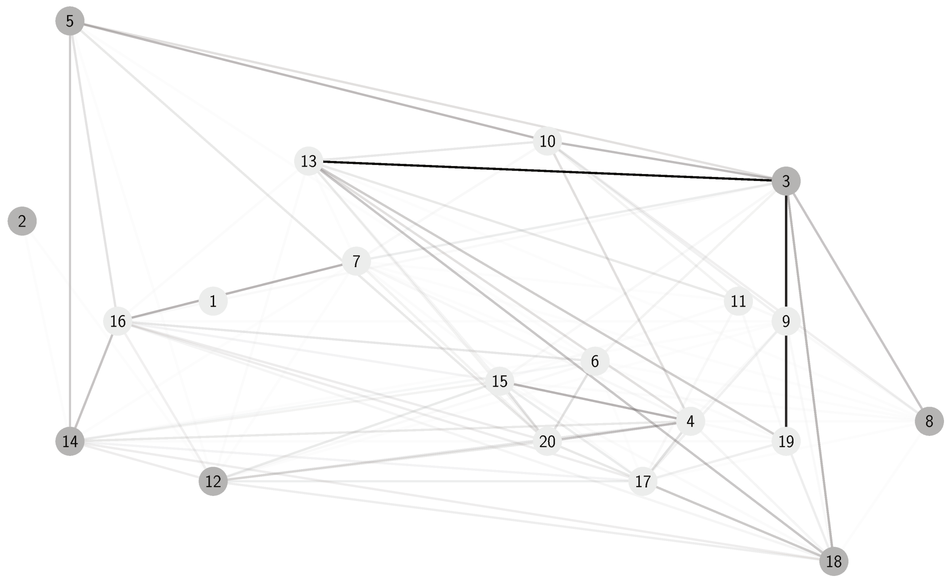


Figure 3.12: Graphical representation of the number of failures generated by each `nocrossing` constraint in a Euclidean TSP instance. The darker the colour of the segment, the higher the number of failures. Darker nodes are those on the boundary of the convex hull.

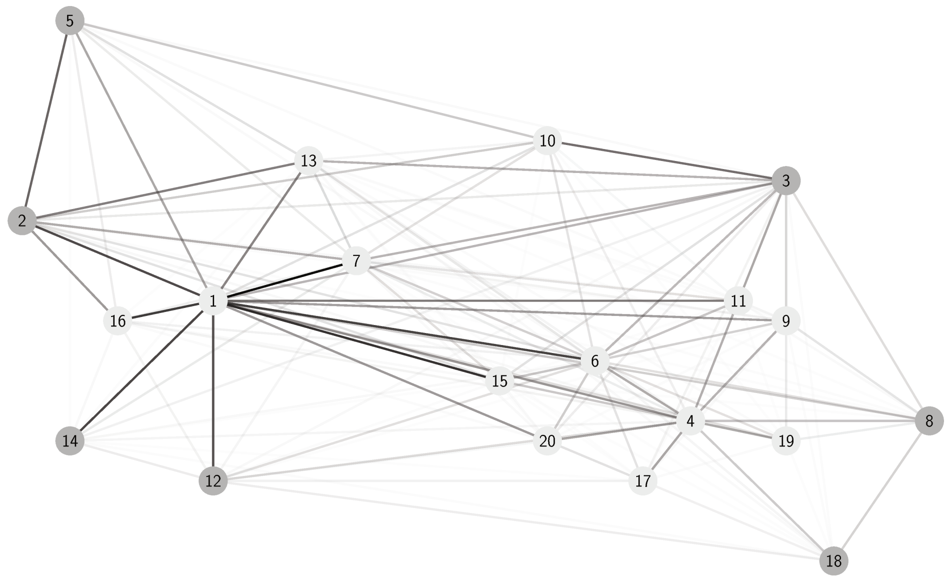


Figure 3.13: Graphical representation of the `RTIO` of each `nocrossing` constraint in a Euclidean TSP instance. The darker the colour of the line, the higher the `RTIO` value. Darker nodes are those on the boundary of the convex hull.

the rigid transformations of the entire point set such as rotations, axis symmetries or even scales.

Because of this, we compute a number of features that try to synthesize some additional information that is invariant with respect to these transformations. As a guideline, we have selected some characteristics that reflect information used by effective TSP solving algorithms, in the hope that they can also serve as a guideline for the effectiveness of the `nocrossing` constraints and in addition we have included features already widely used in the literature to characterise TSPs.

Inspired by works in the literature that attempt to characterise TSP instances by calculating features [119, 160, 146, 106, 186], we decided to introduce features concerning the whole instance rather than the single `nocrossing` constraint. In particular, we focused on a subset of the features introduced by Hutter *et al.* [106] which in turn built on the set previously introduced by Smith-Miles *et al.* [186].

With regard to the algorithms for solving TSPs, the one proposed by Held and Karp [101] based on the spanning tree is particularly effective. The Minimum Spanning Tree (MST) of the node set can be computed in polynomial time and is a popular valid lower bound for the value of the optimal solution. One of the properties we choose for a pair of points is whether the segment connecting them belongs to a minimum spanning tree.

Another interesting property is the so-called necklace condition [64]. Suppose to find a set of discs, each centered on one of the points to be visited, such that the interiors of two discs do not intersect. Clearly, an optimal tour should enter and exit each of the discs, so a valid lower bound is twice the sum of the radii of the discs. From this observation, another interesting property could be the distance of each node to the closest other node.

Finally, in Section 3.3.2 we also introduced constraints that performed pruning based on the convex hull of the set of points; that pruning was also extended to the case of *interior hulls*, after (during search) some of the segments in the current path were already fixed.

Considering what has been introduced so far, we have identified the following 45 features for each `nocrossing`$(i, Next_i, j, Next_j)$ constraint in the dataset. For each instance, let N_i be the point corresponding to P_i with normalized coordinates, so that the coordinates span in the [0,100] interval. Features marked with an asterisk * are those proposed in [106, 186] as introduced previously.

- 1 - 3: **Cost Matrix Statistics***: Mean (c_{avg}), variation coefficient and skew of costs computed between every pair of nodes (points) in the instance.

- 4 - 5: **Distance**: Euclidean distance $d(N_i, N_j)$ between points N_i and N_j and normalized version $d(N_i, N_j)/c_{avg}$, where c_{avg} is the average distance in the cost matrix.

- 6: **Radius***: Mean distance from each node to the instance centroid C. Note that the centroid of a set of points is that point which minimizes the sum of squared Euclidean distances between itself and each point in the set.

- 7 - 10: **Centroid Distance**: Euclidean distance from the instance centroid C to the two extremes N_i ($d(C, N_i)$) and N_j ($d(C, N_j)$) respectively and their normalized versions $d(C, N_i)/c_{avg}$, $d(C, N_j)/c_{avg}$.

- 11 - 12: **Levels**: Level of points P_i and P_j. The idea is to distinguish the points on the perimeter of the convex hull from the internal ones, and have a numeric value suggesting how deep in the interior of the figure is each point. The level of a point P is defined inductively with respect to the set $\mathcal{P}$ of all the points: $lev(P) = lev_{\mathcal{P}}(P)$. The level of a point P with respect to a set $\mathcal{X}$ is 1 if P belongs to the *"exterior"* of $\mathcal{X}$ (precisely, the perimeter $Hull_{\mathcal{X}}$ of the convex hull of $\mathcal{X}$) and is defined inductively as 1 plus the level of P on the *"interior"* set $\mathcal{X} \setminus Hull_{\mathcal{X}}$ otherwise:

$$lev_{\mathcal{X}}(P) = \begin{cases} 1 & \text{if } P \in Hull_{\mathcal{X}} \\ 1 + lev_{\mathcal{X} \setminus Hull_{\mathcal{X}}}(P) & \text{otherwise} \end{cases}$$

- 13 - 15: **Cluster Distance Features***: Mean, variation coefficient, skew of the cluster distance calculated between every pair of nodes in the instance. Cluster distance between a pair of nodes is defined as the minimum bottleneck cost of any path between them in a MST where the bottleneck cost of a path is defined as the largest cost along the path.

- 16 - 17: **Nearest Neighbour Distance***: Standard deviation and coefficient variation of the normalized nearest neighbour distances;

- 18 - 21: **Neighbours Distance**: Euclidean distance of the closest point $C(N_i)$ to N_i ($d(C(N_i), N_i)$) and N_j ($d(C(N_j), P_j)$) respectively, and normalized versions $d(C(N_i), N_i)/c_{avg}$, $d(C(N_j), N_j)/c_{avg}$;

- 22: **Neighbourhood size**: Number of points contained in the circle having as diameter the segment connecting points P_i and P_j;

- 23 - 26: **Minimum spanning tree cost statistics***: Sum, mean, variation coefficient and skew of the edge costs in a minimum spanning tree constructed over all nodes in the instance.

- 27 - 28: **Shortest Path in MST**: Cost of the shortest path p between N_i and N_j in a minimum spanning tree, and its normalized version

- 29 - 31: **Minimum spanning tree node degree statistics***: Mean, variation coefficient and skew of node degrees in a minimum spanning tree constructed over all nodes in the instance.

- 32 - 33: **MST Degree**: Degree in the minimum spanning tree of P_i and P_j respectively;
- 34: **Segment in MST** (boolean value) Indicates whether the segment $\overline{P_iP_j}$ belongs to a minimum spanning tree.
- 35 - 36: **Segment crosses**: Number of segments crossing the segment $\overline{P_iP_j}$ and the version normalized, obtained dividing by the total number of arcs;
- 37 - 40: **Crossings**: Total number of crossings between edges exiting from P_i (resp. P_j) and other edges, and their normalized versions obtained dividing by the total number of arcs.
- 41 - 44: **Fraction of distinct distances***, with precision to $k \in \{1, 2, 3, 4\}$ decimal digits. Each element of the distance matrix is rounded to k decimal digits, creating D^k, then the number of distinct values in such matrix is computed, and divided by the number of values in the matrix;
- 45: **Good**: (Boolean value) label each constraint as useful (1) or useless (0).

3.4.3 Supervised machine learning approaches

Machine learning is a subfield of computer science concerned with developing algorithms capable of improving their performance from experience. The process relies on data, such as examples, or observations, such as direct experience or instruction, to build a statistical model used to solve a practical problem. Examples can be data collected in the (natural) environment, handcrafted by humans, or generated by some other algorithm. Learning can be *supervised*, *semi-supervised*, *unsupervised* and *reinforcement*.

In *supervised learning*, the goal of the algorithm is to use a dataset of labelled examples $\mathbf{X}\{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ to produce a model that takes a feature vector $\boldsymbol{x}$ as input and, outputs information which makes it possible to derive the label for $\boldsymbol{x}$. The feature vector is a vector in which each dimension contains a value $x^{(j)}$, called feature, that describes the example in some way. For all examples in the dataset, the feature at position j in the feature vector always contains the same information type.

The label y_i can be an element belonging to a finite set of classes $\{1, 2, \ldots, C\}$, or a real number, or a more complex structure. If the set of classes consists of only two elements (e.g. $\{0, 1\}$ or $\{positive, negative\}$), the associated machine learning model is called a binary classifier.

The dataset of labelled constraints, presented in the previous section, is suitable for the application of supervised machine learning algorithms, with the goal of learning a model capable of predicting which of the constraints are useful and which are useless in a previously unseen instance of the Euclidean TSP. Each `nocrossing` constraint is labelled as useful or useless so the associated machine learning model results in a binary classifier capable of discriminating between the negative class (*useless*, with label equal to 0) and the positive class (*useful*, with value 1 as label). Among the classification techniques, we consider two well-known and extensively used approaches: Random Forest (RF), known for its good computational performance and scalability, and Neural Network (NN), which have proven to be very effective at modelling correlations among many features.

We now begin our brief description of classification techniques by introducing decision trees as core elements for implementing random forests.

Decision Tree

A *decision tree* classifier (or classification tree) is a predictive model based, as its name suggests, on a tree data structure. Given a trained decision tree, to predict the class of an example, given its feature vector $\boldsymbol{x}$, the tree is traversed from the root to one of the leaf nodes containing the predicted label.

At each branching node a certain feature j of the feature vector $\boldsymbol{x}$ is examined. The result of the test on feature j determines which of the branches will be selected.

Popular algorithms for learning decision trees are CART [43], ID3 [166] and its extension C4.5 [167]. All mentioned algorithms attempt to build the tree by placing tests on attributes that carry more information (information entropy) on the nodes closest to the root; this usually leads to shorter trees that are preferred to longer ones.

Despite of their simplicity in terms of interpretation and implementation, decision trees also have a number of well-known disadvantages: the trees generated tend to be very complex and, if adequate precautions are not adopted, they do not generalise the data well (overfitting); they are not able to manage unbalanced training datasets in an adequate manner; they are sensitive to variations in the training data, which even if small, can lead to the generation of very different trees.

This last aspect, which is a weakness for decision trees, is instead exploited as an advantage in random forests.

Random Forest

A Random Forest (RF) is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control overfitting [42].

The key point is the low correlation between the different decision tree classifiers; uncorrelated models are able to produce ensemble predictions that are more accurate than any of the individual predictions. The random forest ensures that the behaviour of each individual tree is not overly correlated by using two different strategies: *bagging* and *feature randomness.*

Decision trees, as introduced above, are very sensitive to the data with which they are trained so to obtain different tree structures it is enough to make slight modifications to the training set. Random forest takes advantage of this by allowing each individual tree to randomly sample from the dataset with replacement, resulting in different trees. This process is known as *bagging* or bootstrap aggregation. Moreover, in a decision tree when the time to split a node comes, all possible features are considered, and the one chosen is the one that produces the greatest separation between observations at the two sub-nodes. In a random forest the *features randomness* principle is used: each tree can pick only from a random subset of features. This forces more variation between trees in the model and results in less correlation.

We used the implementation available in the WEKA[5] [196] workbench for machine learning. Given a training set $\mathbf{X}$ of instances of Euclidean TSP, learning a random forest in WEKA involves the following steps [9]:

(1) Bootstrap samples $\mathbf{B}_i$ for every tree t_i are drawn by randomly selecting pairs of points (the examples) with replacement from $\mathbf{X}$ until the sizes of $\mathbf{B}_i$ and $\mathbf{X}$ are equal;

(2) A random subset of features (attributes) is selected for each $\mathbf{B}_i$ and used for the training of tree t_i in the forest;

(3) An information gain metric is used to grow unpruned decision trees;

(4) The final classification result is the most popular of the individual tree predictions.

Neural Network

Neural Networks (NNs) are learning models whose structure was inspired by the human brain. NNs are able to identify and model complex interactions among the entities to be classified, in this case constraints.

There are many types of NNs that have different architectures. The simplest one is the Multi-Layer Perceptron (MLP), also called Fully Connected NN [96, 3]. A MLP can be viewed as a graph. It is divided into *layers* and each layer contains nodes, called *neurons*, with each neuron connected to each neuron in the next layer. Each connection is assigned a trainable weight. Every layer has a weights matrix composed of the weights associated with the connections between the current layer and its predecessor. The values of the weights are trained by means of an optimization algorithm, such as gradient descent [47], by computing the gradient of the output of the network with respect to each layer and updating the weights by moving along the gradient.

[5]`https://www.cs.waikato.ac.nz/ml/weka/`

The input data is multiplied by a weights matrix and passed to the neurons of the first layer. The input of each neuron is given to a function, called *activation function*, that calculates the output of the neuron. The output of the set of neurons is provided to the second layer that will use its weights matrix and activation functions to compute the input of the third layer, and so on up to the output layer. This is the last layer of the network and output classification of the given input in the form of a probability distribution among the classes contained in the data. For each class, the MLP outputs a value in the interval $[0, 1]$ and these values add up to 1. Therefore, given the training set, a network must be designed by choosing the number and type of layers, neurons, and activation functions.

The values of the weights are trained by means of *back-propagation* [180]. Given a set of labelled entities, these are passed to the MLP and its output is compared with the correct labels of the entities to compute a real number telling how much the network is able to correctly classify the entities. This value, computed by an error function, is used by an optimization algorithm, such as gradient descent, to tune the weights of the MLP by computing the gradient of the function with respect to each layer and changing the weights by moving along the gradient. These operations are performed iteratively until the error decreases. If the error cannot be further reduced or if its change falls below a certain threshold value through iterations, the training can be stopped.

Machine learning the goodness of constraint propagators

Random Forests and Neural Networks classifiers could be used independently to predict whether the `nocrossing` constraint should be imposed on previously unseen pairs of points, at the same time indicating to avoid imposing constraints that have been assigned the negative class (*useless*).

The whole proposed procedure is the following:

1. A training dataset of various Euclidean TSP instances, where each constraint has been labelled according to its own `RTIO`, is used to learn a RF or MLP classifier;

2. Given any new instance of the problem, for whose constraints the three indicators (and so the class) are unknown, apply the classifier to find pairs of points that are classified as *useful* for imposing the `nocrossing` constraint;

3. Run the instance together with the selected `nocrossing` constraints to solve the Euclidean TSP.

With step 3 we try to eliminate the temporal overhead that could be introduced when searching for a solution due to the constraints recognized by the machine learning model as ineffective (*useless*). One advantage of this method is that,

Table 3.2: Example of confusion matrix.

class	positive (predicted)	negative (predicted)
positive (actual)	TP	FN
negative (actual)	FP	TN

once the classifier has been learnt, it can be reused in each new instance of the problem without having to repeat the machine learning step that has only been executed once.

Overview of model performance assessment

After the model is built on the labelled dataset $\mathbf{X}$, also known as *training set*, the *test set* is used to assess the actual performance of the model. The test set contains examples previously unseen for the learning algorithm. So, if the model is able to correctly predict the labels of the examples in the test set, we can assume that it will also work well when applied to "real cases".

The most commonly used metrics relating to binary classifiers to evaluate the classification model are: confusion matrix, precision/recall, accuracy, F-measure, and ROC AUC (ROC Area)/PR AUC (PR Area).

The *confusion matrix* is a table that summarizes the success of the classification model in predicting examples of different classes.

True Positive TP (respectively Negative TN) is the number of positive (resp. negative) examples correctly classified as positive (resp. negative). False Positive FP (resp. Negative FN) is the number of negative (resp. positive) examples incorrectly classified as positive (resp. negative). True Positive Rate (TP Rate), the proportion of positive examples predicted correctly, and False Positive Rate (FP Rate), the proportion of negative examples predicted incorrectly, are respectively defined as:

$$\text{TP Rate} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad \text{FP Rate} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

Confusion matrices can be used to calculate two important performance metrics: *precision* and *recall*. Precision is the ratio of correct positive predictions to the overall number of positive predictions (defined as the TP Rate):

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Recall is the ratio of correct positive predictions to the overall number of positive examples in the test set:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Accuracy is given by the number of correctly classified examples divided by the total number of classified examples.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

F-measure is the harmonic mean of Precision and Recall.

$$\text{F-measure} = \frac{2}{\text{Precision}^{-1} + \text{Recall}^{-1}}$$

The ROC curve (Receiver Operating Characteristic) and the PR curve (Precision-Recall) are commonly used methods to assess the performance of classification models. The ROC curve plots the TP Rate versus the FP Rate, the PR curve plots the Precision versus the Recall [55, 162].

The highest possible value for all metrics is 1, and, except for the FP Rate, the higher the value, the better the performance.

3.4.4 Experimental evaluation

Machine Learning

The machine learning task was carried out by training, independently, a RF classifier and a MLP classifier.

For the former, WEKA version 3.8.5 [196] was used, for the latter we used Tensorflow[6] [1] version 2.0.0 of the framework with CUDA version 10.0.

The training phase of the RF classifier was controlled by the parameters:

- `-P 100`: size of each bag, as a percentage of the training set size; the default value of 100 was kept;
- `-attribute-importance`: compute and output attribute importance (with the mean impurity decrease method) reported in Figure 3.14;
- `-I 100`: number of iterations (i.e., the number of trees in the random forest);
- `-num-slots 1`: number of execution slots (thread) for constructing the ensemble. The default 1 means no parallelism;
- `-K 0`: sets the number of randomly chosen attributes;
- `-M 1`: the minimum number of instances per leaf;
- `-V 0.001`: minimum numeric class variance proportion of train variance for split (it was kept the default value);
- `-S 1`: seed for random number generator (it was kept the default value).

As for the MLP, we performed a random search to find the best combination for the number of layers, the number of neurons per layer, the optimizer among Adam and Stochastic Gradient Descent, the parameters for the optimizers, and

[6]`https://www.tensorflow.org/`

Table 3.3: Performance of the RF and MLP classifiers over the fraction of instances used for test. Class 0 corresponds to *useless* constraints.

Classifier	Class	Recall	FP Rate	Precision	F-Measure	ROC Area	PR Area	Accuracy
RF	0	0.760	0.155	0.830	0.793	0.883	0.894	
	1	0.845	0.240	0.779	0.811	0.883	0.862	
	W.Avg.	0.802	0.198	0.804	0.802	0.883	0.878	0.802
MLP	0	0.827	0.258	0.704	0.761	0.855	0.871	
	1	0.742	0.173	0.853	0.794	0.855	0.829	
	W.Avg.	0.785	0.215	0.779	0.777	0.855	0.850	0.779

the dropout rate. Prior to training, a standardisation step was carried out on the features. We trained among 300 configurations, and we selected the MLP that showed the best trade-off between classification performance and resource consumption to keep the entire resolution flow of the Euclidean TSP as efficient as possible. The resulting network has 13 layers:

- 2 layers $\times$ 128 neurons,
- 2 layers $\times$ 256 neurons,
- 4 layers $\times$ 1024 neurons,
- 2 layers $\times$ 256 neurons,
- 2 layers $\times$ 128 neurons,
- 1 layer $\times$ 1 neuron.

All the layers, except the last one, use the ReLU activation function, while the last one uses the sigmoid function. Layers 2 to 11 apply batch normalization. Training was done by the Adam optimizer [120] and applying early stopping and learning rate decay.

From the results shown in Table 3.3 it becomes clear that the RF model can classify useful constraints better than MLP model, which achieves a higher score only for recall in classifying useless constraints (class 0).

Euclidean TSP Solver

As already introduced in Section 3.4.2, we decided to empirically evaluate the best threshold value for $\theta^{\texttt{RTIO}}$ to label each constraint as *useful* or *useless*. We solved 192 instances of Euclidean TSP by varying the threshold value, in steps of 0.1, in the interval $[0.1, 2]$. The graph in Figure 3.15 summarizes the results obtained, for each threshold value it shows the geometric mean of the solving time of all 192 instances. The curve shows a rather flat trend up to point $\theta^{\texttt{RTIO}} = 0.6$, which we consider as the minimum and consequently is the value we then used in creating the training dataset for the machine learning models.

To evaluate the improvements in solving time achieved thanks to the predictions performed by the machine learning step, we developed a series of experiments based on randomly generated TSPs. The same techniques described in Section 3.4.2 were

```
0.44 (261828)  NORM-Euclid_Distance_AB [4]
0.43 (248150)  NORM-Euclid_Distance_AB_to_AVG [5]
0.43 ( 80856)  FEAT-Level_PointA [11]
0.41 (223177)  NORM-Distance_A_Centroid [7]
0.41 ( 75512)  FEAT-Level_PointB [12]
0.41 (155679)  INST-Cost_Matrix_AVG [1]
0.4  (223749)  NORM-Distance_B_Centroid [8]
0.39 (196113)  NORM-Distance_A_Centroid_to_AVG [9]
0.39 (148578)  INST-Cost_Matrix_RSD [2]
0.38 (142024)  INST-Cost_Matrix_SKEW [3]
0.38 (198761)  NORM-Distance_B_Centroid_to_AVG [10]
0.36 ( 89783)  FEAT-Neighbourhood [22]
0.36 ( 47483)  FEAT-MST_Degree_A [32]
0.36 (  5282)  FEAT-A_B_Near_In_MST [34]
0.35 (128143)  INST-Radius [6]
0.35 ( 49008)  FEAT-MST_Degree_B [33]
0.34 (152022)  FEAT-Shortest_Path_AB_Weight [27]
0.33 (170974)  NORM-Dist_NN_PointA [18]
0.33 (139810)  FEAT-AB_Crosses [35]
0.33 (157641)  NORM-Dist_NN_PointA_to_AVG [20]
0.32 (153430)  FEAT-Shortest_Path_AB_Weight_to_AVG [28]
0.31 (139985)  FEAT-AB_Crosses_PCT [36]
0.31 (167203)  NORM-Dist_NN_PointB [19]
0.3  (155033)  NORM-Dist_NN_PointB_to_AVG [21]
0.3  (104346)  INST-Cluster_Distance_AVG [13]
0.29 ( 99264)  INST-Cluster_Distance_RSD [14]
0.29 (130485)  FEAT-FromA_Intersection [37]
0.28 ( 98713)  INST-Cluster_Distance_SKEW [15]
0.28 (127202)  FEAT-FromA_Intersection_PCT [39]
0.28 (127660)  FEAT-FromB_Intersection [38]
0.28 ( 94915)  INST-SD_NNDistance [16]
0.27 (126983)  FEAT-FromB_Intersection_PCT [40]
0.27 ( 88123)  INST-RSD_NNDistance [17]
0.26 ( 17647)  INST-MST_Degree_AVG [29]
0.24 ( 80904)  INST-MST_Length [23]
0.23 ( 71190)  INST-MST_Length_AVG [24]
0.22 ( 79410)  INST-MST_Length_RSD [25]
0.22 ( 55912)  INST-MST_Degree_RSD [30]
0.22 ( 55398)  INST-MST_Degree_SKEW [31]
0.22 ( 75762)  INST-MST_Length_SKEW [26]
0.2  ( 34831)  INST-D3Fraction [43]
0.2  ( 57292)  INST-D2Fraction [42]
0.19 ( 61869)  INST-D1Fraction [41]
0.19 ( 15116)  INST-D4Fraction [44]
```

Figure 3.14: Attribute importance based on average impurity decrease (and number of nodes using that attribute). The feature number as presented in Section 3.4.2 is reported in square brackets.

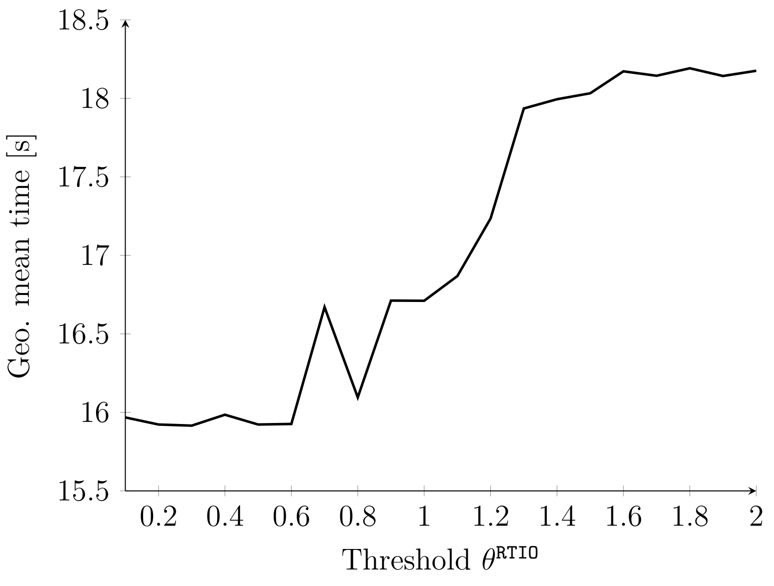


Figure 3.15: Experimental evaluation of the $\theta^{\mathtt{RTIO}}$ value.

used to generate the TSPs used in the experiments. We generated a total of 480 test instances varying the size from 20 to 28 nodes, equally distributed in the three classes *uniform*, *clustered* and *morphed*.

We compared six constraint models based on the successor representation as introduced in Section 3.1.5:

- the basic constraint model described in the preliminaries (denoted as `ECLP`), including the `circuit` and `alldifferent` constraints required by the successor representation plus the objective function;
- the model imposing the `nocrossing` constraints for all pairs of nodes (denoted as `ALL`);
- the model imposing only the `nocrossing` constraints predicted as useful by the RF classifier (denoted as `RF`);
- the model imposing only the `nocrossing` constraints predicted as useful by the MLP classifier (denoted as `MLP`);
- moreover, in order to eliminate the hypothesis that a random removal of constraints could obtain the same speed-up, we also plot the timing results of two constraint models, in which, respectively, 70% of the `nocrossing` constraints were not imposed (denoted as `RAND70`) and half of these constraints were not imposed (denoted as `RAND50`).

All experiments use the *max regret* search strategy [46]. All algorithms are implemented in the ECL^{i}PS^e CLP language [184]. All tests were run on ECL^{i}PS^e

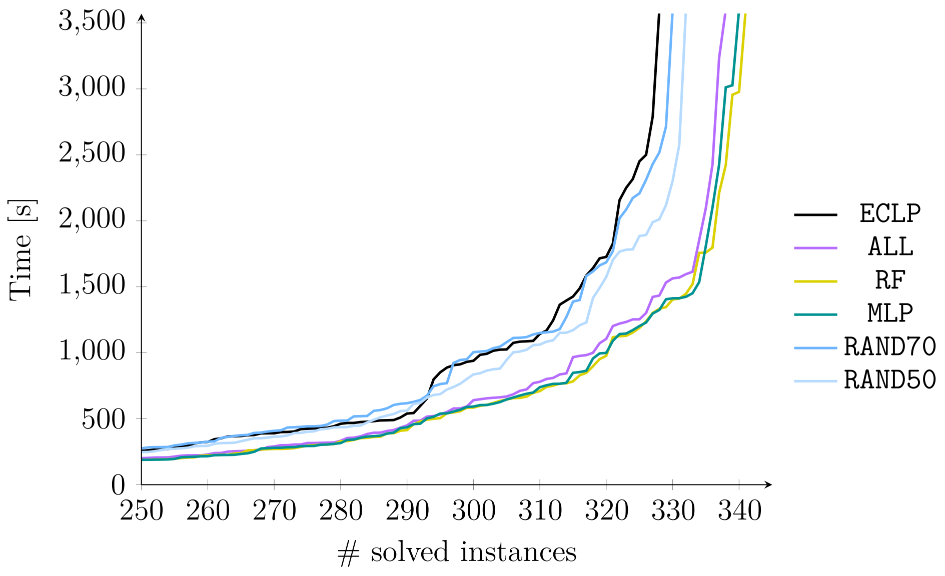


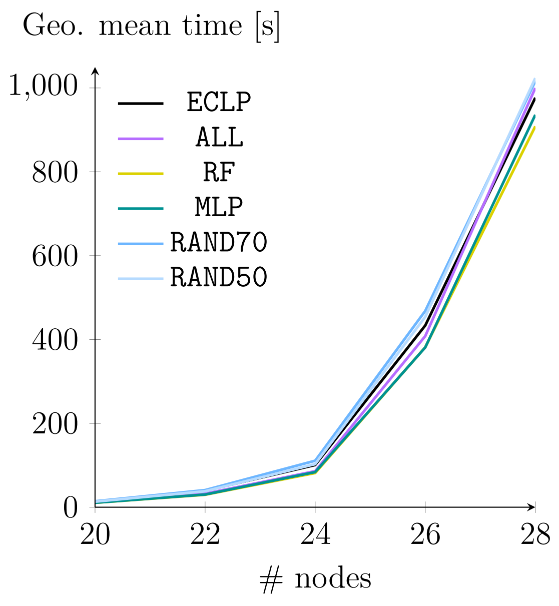
Figure 3.16: Solving time as a function of the number of solved instances. Time limit is 3,600s. Different curves correspond to the different models that were compared during the experiments.

v. 7.0, build #54, on Intel® Xeon® E5-2697 v4 CPUs running at 2.30GHz, with one core and with 1GB of reserved memory. The time limit was 3,600s.

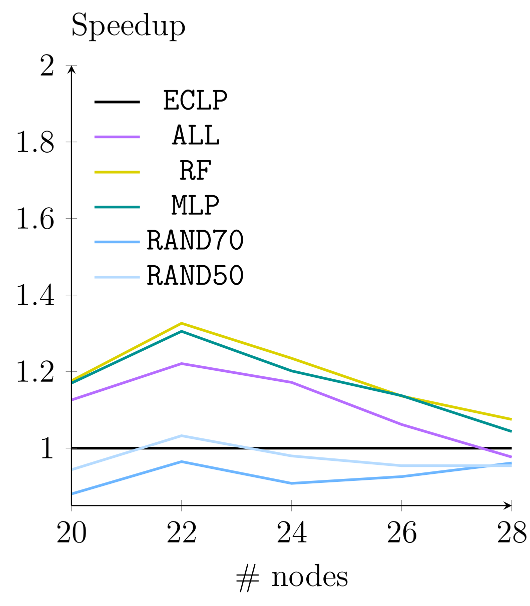
Figure 3.16 shows, for each constraint model, the number of instances that were solved within the timeout. We can see that the basic constraint model `ECLP` is the least effective, while adding `nocrossing` constraints can provide a significant speed-up. Removing constraints randomly is not effective, and it seems that as we increase the number of `nocrossing` constraints from 0 (`ECLP`) to 30% (`RAND70`), to 50% (`RAND50`), to 100% (`ALL`) we obtain increasing speed-ups.

The same does not hold if the constraints to be imposed are carefully selected: the two constraint models in which the `nocrossing` constraints imposed are those predicted through machine learning are the most effective, confirming the effectiveness of the machine learning - based approach proposed. MLP and RF performances are almost overlapping, with RF slightly more effective than the MLP, as expected from the results of Table 3.3, at the cost of higher memory needed: RF requires a memory amount larger one order of magnitude than MLP.

Figures 3.17 and 3.18 show the running time when instance size is varied. It is worth noting that randomly choosing a set of `nocrossing` constraints to be removed does not pay off, and may even increase the running time with respect to adding no `nocrossing` constraints. In particular, in larger instances, the median value of the running time coincides with the timeout for the reference constraint model and for the random selection of constraints. Instead, the median value is significantly lower when selecting the constraints to be imposed with machine learning.

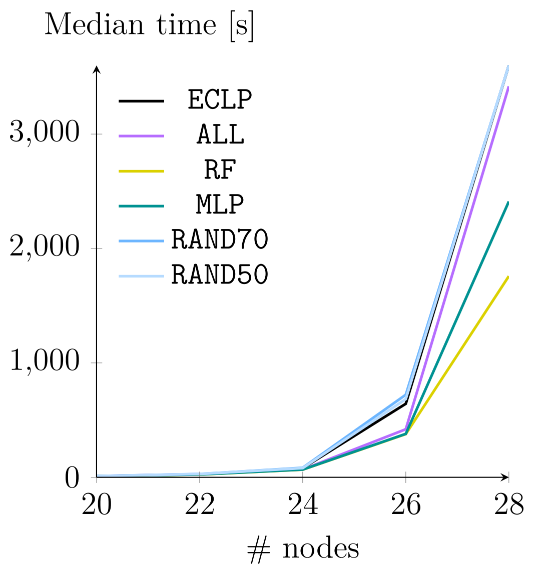


(a)

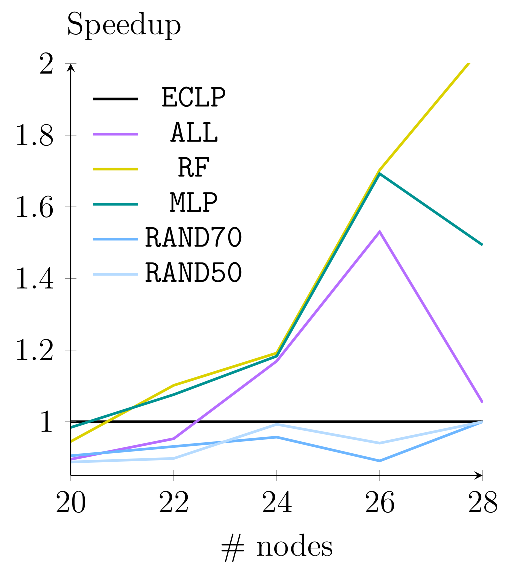


(b)

Figure 3.17: (3.17a) Geometric mean of solving time of TSPs and (3.17b) speedup over the ECLP model, varying the size of the instances.



(a)



(b)

Figure 3.18: (3.18a) Median of solving time of TSPs and (3.18b) speedup over the ECLP model, varying the size of the instances.

The selection of constraints also increased slightly the number of instances that could be solved: among the 480 tested instances, ECLP incurred in timeout on 153 instances, while RAND70 on 151, RAND50 on 149, ALL on 143, RF on 140 and MLP on 141.

4
Branching Interval Algebra

Contents

In the context of automatic reasoning, temporal reasoning is certainly an important field. The applications range from process scheduling to speech recognition, automatic writing, drone swarms coordination, and much more.

One of the most prominent formalisms in this area is *Allen's Interval Algebra* [6], ($\mathcal{IA}$), introduced in 1983, which is based on the basic concepts of *event* and *relation* between events.

$\mathcal{IA}$ is an algebra where relations are *qualitative*, as no metric information is kept about the duration of an event or the absolute time interval between two events. An interesting problem that emerges in this framework is the *consistency problem* for an algebra $\mathcal{A}$ (SAT($\mathcal{A}$)): given a set of event variables and a set of relation-constraints between the variables, the goal is to decide whether all variables can be *modelled* into actual events (interval) without violating the constraints. SAT($\mathcal{IA}$), which is archetypal of the class of temporal constraint satisfaction problems, was proven to be NP-COMPLETE, in the general case, by Vilain and Kautz [192].

In the following years, researchers studied the properties of the $\mathcal{IA}$ to identify its *maximally tractable fragments* (and especially *subalgebras*), and in 2003 Krokhin et al. [124] gave a complete picture of tractability of fragments of the $\mathcal{IA}$ with respect to consistency.

A tree-like extension of the linear formalism was proposed [168], and called *Branching Interval Algebra* ($\mathcal{BA}$). Tree-like temporal reasoning is ubiquitous in computer science, mainly at the logical level (see, for example, the huge amount of work that exists on $\mathcal{CTL}$, $\mathcal{CTL}^*$, and similar formalisms [66, 65]). Since SAT($\mathcal{BA}$) is also NP-COMPLETE, the problem of identifying its tractable fragments arises again in a natural way.

Branching Interval Algebra also has many potential applications in different areas of Artificial Intelligence; for example, in planning with alternatives, in which different plans have to be taken into account in the design phase, or in the automatic generation of narratives and in the formal verification of parallel programs. Consequently, it is important to be able to efficiently solve classical problems expressed in $\mathcal{BA}$. This can be achieved in two steps: first, by identifying sufficiently expressive but tractable fragments of the whole algebra, and, second, by using such fragments to boost the performances of a backtracking algorithm for the whole language.

In the following we study the tractability of fragments of the $\mathcal{BA}$, and we give an almost complete picture of their situation. We identify, in particular, four interesting tractable fragments, and study their maximality both with respect to their tractability and with respect to their *PC-tractability*, that is, their tractability with the algorithm of *Path Consistency*. Moreover, we design an enhanced version of the classic backtracking consistency algorithm for the full $\mathcal{BA}$ that takes advantage from tractable fragments of it, and perform a series of experiments to test its applicability.

The content of this chapter has been arranged and adapted from some of the author's works that have already been published in conferences and journals, see [31, 33, 32].

4.1 Preliminaries

In the following we will introduce and investigate four qualitative temporal algebras: $\mathcal{PA}$ (Point Algebra), $\mathcal{BPA}$ (Branching Point Algebra), $\mathcal{IA}$ (Interval Algebra) and

$\mathcal{BA}$ (Branching Algebra); these are all *Tarski relation algebras* [125] where the *atoms* are the *pairwise disjoint* and *jointly exhaustive* basic relations, generically denoted with $r, s, \ldots$, and so on. A set of disjunctions between atoms $\{r_1, \ldots, r_n\} = r_1 \vee \cdots \vee r_n$ is a *relation*; relations are generically denoted with $R, S, \ldots$, and so on. We express the fact that two objects, e.g. two intervals, X and Y are related by R with XRY. Given a set of basic relations $\mathcal{A}_{basic}$, we can build the relation algebra $\mathcal{A}$ as the powerset $2^{\mathcal{A}_{basic}}$. To stress the algebra $\mathcal{A}$ over which we are considering a certain relation R, we use the notation $R_{\mathcal{A}}$. Among all the relations in $\mathcal{A}$, three are particularly important: the empty relation $\bot = \emptyset$, the *all* (or *unknown*) relation $\top = \mathcal{A}_{basic}$, and the *identity relation* $\equiv$, which is $\{=\}$ in $\mathcal{PA}$, $\mathcal{BPA}$ and their fragments, and $\{e\}$ in $\mathcal{IA}$, $\mathcal{BA}$, and their fragments. A relation algebra is also equipped with the unary operations *complement* (denoted by $^\neg$) and *converse* ($^\smile$),

$$\forall x, y, R \colon x(R^\neg)y \iff x(\top \setminus R)y$$
$$\forall x, y, R \colon x(R^\smile)y \iff y(R)x$$

the binary operations *union* ($\cup$), *intersection* ($\cap$),

$$\forall x, y, R, S \colon x(R \cup S)y \iff x(R)y \vee x(S)y$$
$$\forall x, y, R, S \colon x(R \cap S)y \iff x(R)y \wedge x(S)y$$

and *weak composition* ($\diamond$), defined as:

$$r \diamond s = \{t \mid \exists x, y, z \colon x(r)z \wedge z(s)y \Rightarrow x(t)y\}$$

$$R \diamond S = \bigcup_{r \in R, s \in S} r \diamond s$$

In temporal algebras, weak composition is mostly referred to as, simply, *composition* (see, e.g., [6, 192, 124, 168, 191, 154, 132, 63]). Yet, in more recent publications (see, e.g., [175, 8]), a more precise distinction between weak composition and (real) composition was made, and (real) composition was defined as:

$$R \circ S = \{(x, y) \mid \exists z \colon x(r)z \wedge z(s)y\}$$

The composition of two relations R and S represents the *strongest implied relation* between two events connected to a third one, and it is, in a sense, the application of the transitive property (see, e.g. [174]). Unfortunately, (real) composition is basically useless because it depends on the realization of the variables, and in temporal algebras the domain of the variables is infinite and dense. Observe, also, that in our notation we use $\diamond$ for weak composition and $\circ$ for composition, as in [154], but unlike other authors do. Finally, we define the *strong composition* ($\bullet$) of three relations as:

$$x(\bullet(R, S, T))y \iff^{def} x(R \diamond S)y \wedge x(T)y$$

The strong composition takes into account a previous information already existing between the two events.

4.1.1 The Point Algebra and the Interval Algebra

In Point Temporal Algebra $\mathcal{PA}$, events are represented as points on the timeline. Variables over time points are denoted with $x, y, \ldots$, and so on. Point Temporal Algebra was historically introduced only later than Allen's Interval Algebra where events are instead represented as intervals. $\mathcal{PA}$ is based on the notion of relations between pairs of variables, the set of basic point relations ($\mathcal{PA}_{basic}$) consists of three basic relations $\{=, <, >\}$ (referred also as a linear model of time), where $<$ is the total order relation, $>$ its converse and $=$ the equivalence relation.

Definition 4.1.1 (Point Algebra $\mathcal{PA}$)**.** $\mathcal{PA}$ is the algebraic structure with underlying powerset $P(\mathcal{PA}_{basic})$, with the operations of converse, complement, intersection, union, weak composition and strong composition, and the following axioms hold:

- $\forall x : (x = x)$ (reflexivity)
- $\forall x, y : (x = y) \Rightarrow (y = x)$ (symmetry)
- $\forall x, y, z : (x = y) \wedge (y = z) \Rightarrow (x = z)$ (transitivity)
- $\forall x, y : (x < y) \Rightarrow (x \neq y)$ (incompatibility)
- $\forall x, y : (x < y) \Rightarrow (y > x)$ (converse)
- $\forall x, y, z : (x < y) \wedge (y < z) \Rightarrow (x < z)$ (transitivity)
- $\forall x, y : (x = y) \vee (x < y) \vee (y < x)$ (total order)

We often use a classical abbreviation for relations, for example $\{<, =\}$ is written as $\leq$, $\perp = \emptyset$ denotes the empty relation, while $\top = \{<, =, >\}$ denotes the union of all basic relations. We remind that $|\mathcal{PA}_{basic}| = 3$ so $|\mathcal{PA}| = 2^3 = 8$ relations, by enumeration, we have: $\mathcal{PA} = \{\perp, <, \leq, =, \neq, \geq, >, \top\}$.

As mentioned above, extending the point algebra to intervals results in the Interval Algebra ($\mathcal{IA}$) originally introduced by J.F. Allen in 1983.

In Allen's $\mathcal{IA}$ an event is a 2-*interval*, that is, an interval defined by a starting and an ending point over a total order, while a *relation* is a set of disjunctions between *basic relations*, which represent the possible positions that two intervals can assume relatively to each other.

We generally denote intervals with $I, J, \ldots$, their *starting* endpoints with I^-, $J^-, \ldots$, and so on, their *finishing* endpoints with $I^+, J^+, \ldots$, and so on, and if we wish to refer both endpoints at the same time we use $I^\pm, J^\pm, \ldots$, and so on.

Variables over intervals are denoted with $X, Y, \dots$ In the linear model of time, one can describe 13 *basic interval relations* ($\mathcal{IA}_{basic}$) of the (linear) *Interval Algebra*, which are derived by the possible positions that the endpoints of two intervals can assume relatively to each other: these relations are *before (b), meets (m), overlaps (o), starts (s), during (d), finishes (f)*, their converses *after (bi), met-by (mi), overlapped-by (oi), started-by (si), contains (di), finished-by (fi)* and the self-symmetric *equals (e)* depicted in Figure 4.1.

Definition 4.1.2 (Interval Algebra $\mathcal{IA}$). $\mathcal{IA}$ is the algebraic structure with underlying powerset $P(\mathcal{IA}_{basic})$, with the operations of converse, complement, intersection, union, weak composition and strong composition and the following axioms hold:

- $\forall I : (I^- < I^+)$
- $\forall I, J : (I^+ < J^-) \Rightarrow (IbJ)$
- $\forall I, J : (I^+ = J^-) \Rightarrow (ImJ)$
- $\forall I, J : (I^- < J^-) \wedge (J^- < I^+) \wedge (I^+ < J^+) \Rightarrow (IoJ)$
- $\forall I, J : (I^- = J^-) \wedge (I^+ < J^+) \Rightarrow (IsJ)$
- $\forall I, J : (J^- < I^-) \wedge (I^+ < J^+) \Rightarrow (IdJ)$
- $\forall I, J : (J^- < I^-) \wedge (I^+ = J^+) \Rightarrow (IfJ)$
- $\forall I, J : (I^- = J^-) \wedge (I^+ = J^+) \Rightarrow (IeJ)$
- $\forall I, J, r : (IrJ) \Leftrightarrow (JriI)$

Since $|\mathcal{IA}_{basic}| = 13$ then $|\mathcal{IA}| = 2^{13}$, and like in $\mathcal{PA}$, $\perp = \emptyset$ denotes the empty relation, while $\top$ denotes the union of all basic relations.

4.1.2 The Branching Point Algebra and the Branching Interval Algebra

Later, $\mathcal{PA}$ and $\mathcal{IA}$ were extended to the branching-time case, leading to the introduction of Branching Point Algebra ($\mathcal{BPA}$) [171] and Branching Interval Algebra ($\mathcal{BA}$) [168].

In branching-time algebras, events are modelled in a tree-like structure rather than on a line. Let $(\mathcal{T}, <)$, or simply $\mathcal{T}$, be a partial order, whose elements (points) are generally denoted by $p, q, \dots$, and so on, and where $p \parallel q$, which is equal to $p \not< q \wedge p \neq q \wedge p \not> q$, denotes that p and q are incomparable with respect to the ordering relation $<$. A partial order $\mathcal{T}$ becomes a *right tree order* if it always holds that:

$$(x \parallel y) \wedge (y < z) \Rightarrow (x \parallel z)$$

Such a tree order is called a *future branching model of time*, and there are four relations which may hold between any two points: *equals* ($=$), *incomparable* ($\parallel$), which are symmetric, and *before* ($<$) and *after* ($>$), which are converse of each other. These relations are depicted in Figure 4.2, and are called *basic point relations* of the *Branching Point Algebra* ($\mathcal{BPA}$), and together form the set called $\mathcal{BPA}_{basic}$.

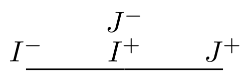
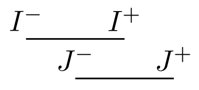
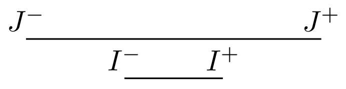
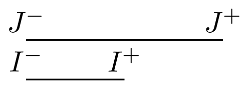
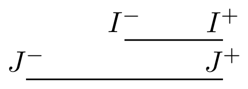
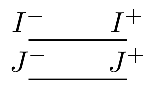
b (bi)	*I before J*	$I^+ < J^-$	
m (mi)	*I meets J*	$I^+ = J^-$	
o (oi)	*I overlaps J*	$I^- < J^- < I^+ < J^+$	
d (di)	*I during J*	$J^- < I^- < I^+ < J^+$	
s (si)	*I starts J*	$I^- = J^- < I^+ < J^+$	
f (fi)	*I finishes J*	$J^- < I^- < I^+ = J^+$	
e	*I equals J*	$I^- = J^- < I^+ = J^+$	

Figure 4.1: A pictorial representation of the thirteen basic interval relations in $\mathcal{IA}$. To obtain the converse relations, it is sufficient to swap I and J.

Definition 4.1.3 (Branching Point Algebra $\mathcal{BPA}$)**.** $\mathcal{BPA}$ is the algebraic structure with underlying powerset $P(\mathcal{BPA}_{basic})$, with the operations of converse, complement, intersection, union, weak composition and strong composition, and the following axioms hold:

- $\forall x : (x = x)$ (reflexivity)
- $\forall x, y : (x = y) \Rightarrow (y = x)$ (symmetry)
- $\forall x, y, z : (x = y) \wedge (y = z) \Rightarrow (x = z)$ (transitivity)
- $\forall x, y : (x < y) \Rightarrow (x \neq y)$ (strict order)
- $\forall x, y : (x < y) \Rightarrow (y > x)$ (converse)
- $\forall x, y, z : (x < y) \wedge (y < z) \Rightarrow (x < z)$ (transitivity)
- $\forall x, y : (x < y) \vee (x > y) \vee (x = y) \vee (x \parallel y)$ (partial order)
- $\forall x, y, z : (x \parallel y) \wedge (y < z) \Rightarrow (x \parallel z)$ (tree-order)
- $\forall x, y, z : (x \parallel y) \wedge (y = z) \Rightarrow (x \parallel z)$ (tree-order)

A way to extend the linear basic interval relations to the branching case was suggested by Ragni and Wölfl [168]. By taking into account the *splitting points*, i.e., the points where two branches start diverging, in addition to the endpoints of the intervals, gives a total of 24 basic relations, some of which can only be distinguished through first-order quantification, at least if one does not want to explicitly embed the splitting points in the system. For example, in Figure 4.3 we see that, to distinguish the two situations, we need to quantify of the existence, or non-existence, of a point between a and c. To overcome this problem, that becomes

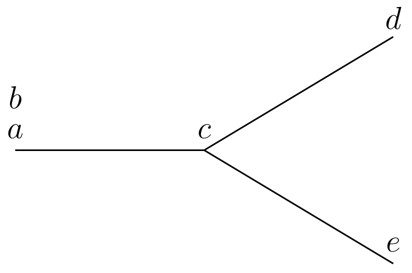


Figure 4.2: A pictorial representation of the four basic branching point relations, where $a = b$, $a < c$, $d > c$, and $d \parallel e$

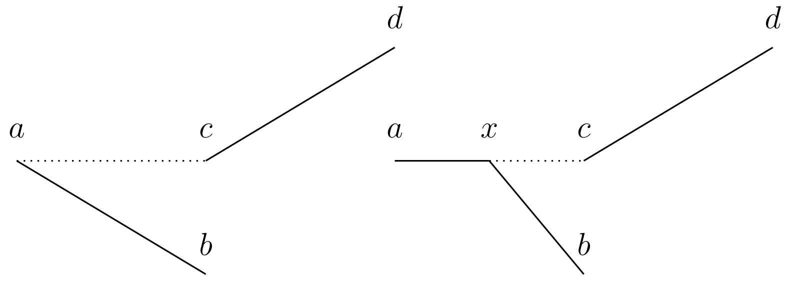


Figure 4.3: An example of two situations that require quantification to be distinguished in the language of endpoints.

ib (*ibi*)	*I init.before J*	$I^- < J^- \parallel I^+$
im (*imi*)	*I init.meets J*	$I^- < J^- < I^+ \parallel J^+$
ie	*I init.equals J*	$I^- = J^- < I^+ \parallel J^+$
u	*I unrelated J*	$I^- \parallel J^-$

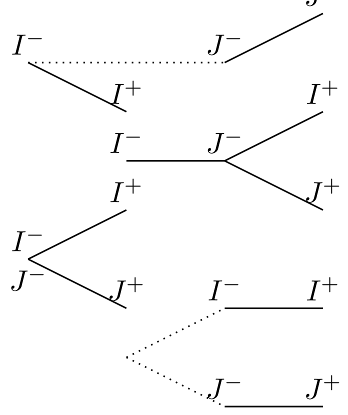


Figure 4.4: A pictorial representation of the six basic branching interval relations. Solid lines are actual intervals, dashed lines complete the underlying tree structure.

relevant when we study the behaviour of branching relations in association with the behaviour of branching point relations (that is, by studying the properties of their point-based translations), Ragni and Wölfl also introduce a set of coarser relations, characterized by being translatable to point-based relations using only the language of endpoints, without quantification. That adds only six new basic relations to the linear ones, to describe which no first-order quantification is needed: these are *initially before (ib)*, *initially meets (im)*, their converses *initially after (ibi), initially met-by (imi)* and the self-symmetric *unrelated (u)* and *initially equals (ie)* depicted in Figure 4.4. This new set of 19 basic relations is the $\mathcal{BA}_{basic}$ set of the *Branching Interval Algebra* $\mathcal{BA}$.

Definition 4.1.4 (Branching Interval Algebra $\mathcal{BA}$)**.** $\mathcal{BA}$ is the algebraic structure with underlying powerset $P(\mathcal{BA}_{basic})$, and with the operations of converse, complement, intersection, union, weak composition and strong composition, and the $\mathcal{IA}_{basic}$ axioms hold, plus the following ones:

- $\forall I, J : (I^- \parallel J^-) \Rightarrow (IuJ)$
- $\forall I, J : (I^- = J^-) \wedge (I^+ \parallel J^+) \Rightarrow (IieJ)$
- $\forall I, J : (I^- < J^-) \wedge (I^+ \parallel J^-) \Rightarrow (IibJ)$
- $\forall I, J : (I^- < J^-) \wedge (I^+ \parallel J^+) \Rightarrow (IimJ)$
- $\forall I, J : (I^- > J^-) \wedge (I^- \parallel J^+) \Rightarrow (IibiJ)$
- $\forall I, J : (I^- > J^-) \wedge (I^+ \parallel j^+) \Rightarrow (IimiJ)$

4.1.3 Point-mapping Operator and Closure of a Relation Algebra

The weak composition of basic relations is usually computed by using a *weak composition table*; the one for the $\mathcal{PA}$ is shown in Table 4.1 while the one for the $\mathcal{BPA}$ is shown in Table 4.2.

Although Allen explicitly wrote the 13-by-13 weak composition table of the $\mathcal{IA}$, and the same did other authors for other algebras, this process is quite annoying and error-prone, especially for bigger interval algebras such as $\mathcal{BA}$, where a 19-by-19 table would be necessary. We can, instead, define a *point-mapping operator* ξ, which transforms interval relations to a set of disjunctions of conjunctions of point relations:

$$\xi(r) = \{r^-, r^\mp, r^\pm, r^+\}$$

where:

$$I(r)J \overset{def}{\iff} I^-(r^-)J^- \wedge I^-(r^\mp)J^+ \wedge I^+(r^\pm)J^- \wedge I^+(r^+)J^+$$

In other words, ξ returns the set of four-point relations that link the endpoints of two intervals when they are related via a certain interval relation; $r^\mp$, for example,

$\diamond$	$<$	$>$	$=$
$<$	$\{<\}$	$\top_{\mathcal{PA}}$	$\{<\}$
$>$	$\top_{\mathcal{BPA}}$	$\{>\}$	$\{>\}$
$=$	$\{<\}$	$\{>\}$	$\{=\}$

Table 4.1: Weak composition table of $\mathcal{PA}_{basic}$ relations.

$\diamond$	$<$	$=$	$>$	$\|$
$<$	$\{<\}$	$\{<\}$	$\top_{\mathcal{PA}}$	$<\|$
$=$	$\{<\}$	$\{=\}$	$\{>\}$	$\{\|\}$
$>$	$\top_{\mathcal{BPA}}$	$\{>\}$	$\{>\}$	$\{\|\}$
$\|$	$\{\|\}$	$\{\|\}$	$>\|$	$\top_{\mathcal{BPA}}$

Table 4.2: Weak composition table of $\mathcal{BPA}_{basic}$ relations.

relates the beginning point of I (that is, I^-) with the ending point of J (J^+). This allows us to manually write a weak composition table only for a certain point algebra and then derive the weak composition in the corresponding interval algebra through the point mapping. In particular, for any interval algebra $\mathcal{A}$, we can define the *pointisable* fragment $\mathcal{A}_{point}$ as the fragment of those and only those interval relations R such that $|\xi(R)| = 1$. Instances of $\mathcal{A}_{point}$ relations can be easily reduced in linear time to equivalent instances in the respective point algebra by using ξ. Given a relation algebra $\mathcal{A}$, a set $\mathcal{S} \subseteq \mathcal{A}$, and a set of operations $\mathcal{F} = \{f_1, \ldots, f_n\}$, with arity $\{|f_1|, \ldots, |f_n|\}$, defined over $\mathcal{A}$, the *closure* of $\mathcal{S}$ over $\mathcal{F}$ is given by the following inductive definition:

$$C^1_{\mathcal{F}}(\mathcal{S}) = \{R \mid \forall f \in \mathcal{F}, \forall S^{|f|} \in \mathcal{S}^{|f|} \colon R = f(S^{|f|})\} \cup \mathcal{S}$$
$$C^i_{\mathcal{F}}(\mathcal{S}) = C^1_{\mathcal{F}}(C^{i-1}_{\mathcal{F}}(\mathcal{S}))$$
$$C_{\mathcal{F}}(\mathcal{S}) = C^{|\mathcal{A}|}_{\mathcal{F}}(\mathcal{S})$$

Given a relation algebra $\mathcal{A}$ and a set $\mathcal{S} \subseteq \mathcal{A}$, its *weak closure* is its closure under converse, intersection, and weak composition, and it is denoted by $\overline{\mathcal{S}}$ (that is, $\overline{\mathcal{S}} = C_{\smile,\cap,\diamond}(\mathcal{S})$), while its *strong closure* is its closure under converse, intersection, and strong composition, and it is denoted by $\widehat{\mathcal{S}}$ (that is, ($\widehat{\mathcal{S}} = C_{\smile,\cap,\bullet}(\mathcal{S})$). Clearly, a weak closure is also a strong one; the converse holds if $\mathcal{S}$ contains $\top$. Examples of strong subalgebras that are not weak ones can be found in [154, 94]; as we shall see, interesting fragments of the $\mathcal{BA}$ also display a similar behaviour.

The notion of strong composition and strong closure comes handy when looking for tractable fragments of an algebra: there are fragments which are not closed under weak composition, but if we intersect the resulting relation with any element of the fragment (i.e., we apply strong composition), we can guarantee that we will always obtain a relation inside the fragment.

4.1.4 Problems and Tractability

Several problems can be studied in a relation algebra $\mathcal{A}$, but first we must introduce some notation. An $\mathcal{A}$-*instance* (or, simply, *instance*) Θ is a pair (V, E), where V is a set of variables and $E \subseteq (V \times \mathcal{A} \times V)$ is a set of $\mathcal{A}$-*constraints* (also written simply constraints in this chapter).

Each $\mathcal{A}$-*constraint* is denoted by XRY where R is the relation between the variable $S = \{X, Y\}$ (its scope). Given a constraint XRY, a *sub-constraint* is a constraint $XR'Y$ where $R' \subseteq R$. A constraint is *basic* if it does not have any proper sub-constraints (i.e., if R is a basic relation). An instance can be represented by a multigraph, and it is said to be *normalized* if and only if there is exactly one constraint bewteen any different pair of variables, so its associated multigraph is a complete graph; in this case, the instance is also called a *network*.

Given an instance $\Theta = (V, E)$, a *sub-instance* of Θ is an instance $\Theta' = (V, E')$ where all the constraints in E' are sub-constraints of the constraints in E, while a *projection* of Θ is an instance $\Theta' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ is the set of all an only those edges that insist on vertexes that are both in E'. An *instantiation* of Θ is a sub-instance Θ' where all the constraints are basic; an *interpretation* $\hat{\Theta}$ of Θ is a realization of the variables in V with concrete elements of the domain (e.g., points or intervals), and a *model* (if it exists) is an interpretation which satisfies all the constraints in E; finally, a *solution* of Θ is an instantiation which has at least one model. Any interpretation of an instance Θ corresponds to a unique instantiation of Θ, but while the number of interpretations, and possibly models, is infinite, the number of instantiations, and therefore solutions, is always bounded. Since it is trivial to generate an interpretation from an instantiation, as it is establishing whether an instantiation is a solution, one is usually more interested in finding solutions than actual models.

Given an instance Θ, the *all solution problem* is the problem of finding the set $\boldsymbol{\Theta}$ of solutions of Θ. The *consistency problem* (SAT) is the problem of deciding the existence of a model for Θ. Finally, the *minimality problem* (MIN) is the problem of deciding the minimality of Θ; an instance is said to be *minimal* if it holds that all of its strict sub-instances Θ' give rise to a strict subset of solutions, that is, if $\boldsymbol{\Theta}' \subset \boldsymbol{\Theta}$. Since SAT is polynomially equivalent to MIN [192], researchers mostly focused on SAT when trying to prove the tractability (or intractability) of a certain relation algebra or of one of its fragments. For this reason, we say that a fragment $\mathcal{S} \subseteq \mathcal{A}$ is *tractable* if and only if SAT($\mathcal{S}$) is tractable.

For many temporal algebras, including $\mathcal{IA}$ and $\mathcal{BA}$, the consistency problem for an instance is in NP because there exists a simple non-deterministic algorithm that solves it, which, given $\Theta = (V, E)$, guesses the relative position of $2 \cdot |V|$ points and checks if every constraint is respected, and it is hard for NP [192, 168]; on the practical side, these kind of problems are often solved via popular heuristics such as constraint propagation and local consistency. An instance Θ is said to be

k-consistent if, given any consistent realization of $k-1$ variables, there exists an instantiation of any k-th variable such that the constraints between the subset of k variables can be satisfied together; it is said to be *strongly* k-consistent if it is k'-consistent for every $k' \leq k$ [139], and if an instance is strongly k-consistent, then it must also have minimal labels. Because of the nature of temporal algebras, their instances are always 1-consistent (*node consistent*) and 2-consistent (*arc consistent*), by definition. Enforcing *path consistency*, that is, 3-consistency corresponds to apply the following simple algorithm (*Path Consistency* [139, 150]): for every triple (s,t,z) of variables in $\Theta = (V,E)$ such that $sRt, sR_1z, tR_2z \in E$, replace sRt by $s(R \cap (R_1 \diamond R_2))t$. Clearly, if enforcing path consistency results in at least one empty constraint, Θ is not consistent. In general enforcing path consistency (in fact, k-consistency for any constant k) does not imply consistency; if consistency can be decided by *Path Consistency*, then we say that it is *PC-tractable*.

The tractability by *Path Consistency* is interesting on its own due to the high popularity of this algorithm, and due to the possibility of employing a PC-tractable fragment within the classic backtracking algorithm as a speed-up heuristic in a natural way. A necessary condition for *Path Consistency* to be used on instances whose constraints belong to a fragment $\mathcal{S}$ is that $\mathcal{S}$ itself is closed under converse, intersection, and at least strong composition. Tractability of fragments is usually studied starting by their generators; in the classical literature, where weak composition is generally used, such a strategy is based on the fact that SAT($\overline{\mathcal{S}}$) and SAT($\mathcal{S}$) are polynomially equivalent when $\top, \equiv \in \mathcal{S}$ [154]. In some cases, such as in branching temporal algebras, we encounter fragments which are only closed under strong composition and such that $\top \notin \mathcal{S}$; interestingly enough, the same strategy works in these cases as well, that is, SAT($\widehat{\mathcal{S}}$) and SAT($\mathcal{S}$) are also polynomially equivalent.

Theorem 4.1.1. *For a fragment $\mathcal{S} \subseteq \mathcal{A}$,* SAT*($\widehat{\mathcal{S}}$) and* SAT*($\mathcal{S}$) are polynomially equivalent.*

Proof. The left to right implication is trivial, since $\mathcal{S} \subseteq \widehat{\mathcal{S}}$. For the other direction, we can proceed as in the weak case: it is always possible to replace any relation of the kind $X\ (R^{\smile})\ Y$ with $Y\ (R)\ X$, and any relation of the kind $X\ (\bullet(R,S,T))\ Y$ with $X\ (R)\ Z$, $Z\ (S)\ Y$ and $X\ (T)\ Y$, where Z is a new variable. By applying these transformations up to $|\mathcal{A}|$ times every $\widehat{\mathcal{S}}$-relation is converted into a $\mathcal{S}$-relation. $\square$

4.2 Applying Branching Algebra

Planning with Alternatives

The use of $\mathcal{IA}$, and in particular of $\mathcal{IA}$-networks of constraints, to model planning problems is ubiquitous in the literature (see, e.g. [143, 152, 197]). A typical modelling exercise involves a set of *tasks* to be executed for a *goal* to be reached. Plans modelled with linear time, however, allow no margin for error: once the plan is in place, every task must be executed in a precise way. Using branching time, we can develop plans that have alternative routes that can be taken in case some action fails, or more in general, plans that already consider different outcomes. Let us consider the case of an *automated (smart) home.* In this example there are a number of actions to be planned at the beginning of the week, subject to several constraints, some of which may depend on the weather. We model both the action of *forecast reading* and its possible outcomes *rain* and *dry* as tasks, forcing both of them to take place in the future of the action of reading the forecast, and to be incomparable to each other; they are incomparable, and not just non-overlapping, because they are interpreted as information that hold on the entire week. In this way, we make sure that the planning agent returns a plan, if it exists, with both alternatives, each characterized by the fact that *it will be raining sometimes during the week* or *the weather will be dry during the whole week*; such a plan is not (only) dynamic: rather, it is an account of all actions that will be taken in each case, and, as such, can be further processed if necessary (e.g., we can compute the total energetic cost in each case and make sure that it is within the established limits). The actions that one may want to plan include: *lawn mowing* and *lawn watering* (in case of no rain), *dishwasher activation*, *washing machine activation*, *rooms ventilation* (in case of no rain), *rooms warming* (in case of rain). Typical constraints include that dishwashing, laundry, and lawn mowing are never overlapping (because the instantaneous energy consumption may be too high), and that lawn mowing occurs, if at all, always before watering. A network that includes all constraints can be seen in Figure 4.5.

Automatic Generation of Narrative

Generation of narrative is a modern application of artificial intelligence, specifically of natural language processing [189]. While the classical applications of automatically generated narratives include weather reports, instructions, descriptions of museum artifacts, narratives can be also used as the basis of automatic storytelling and plot generation [176]. Many modern and classic science-fiction stories, movies, and even video games make substantial use of parallel, incomparable timelines. To keep an adequate cause-effect consistency, however, in presence of non-trivial literary *escamotage* (e.g., time travel), modelling the basic elements with $\mathcal{BA}$ may be a solution. The generated narrative can be checked for consistency to ensure that, while possibly non-linear, it is internally coherent.

$$\begin{tikzcd}
 & X_8 \arrow[dl, "\{b,m,mi,bi\}"'] \arrow[dd, "\{b,d,f\}"] & & & X_5 \arrow[dd, "\{b,d,f\}"'] & \\
X_6 \arrow[dr, "\{b,d,f\}"'] & & X_7 \arrow[dl, "\{b,d,f\}"] & X_9 \arrow[dr, "\{b,d,f\}"'] & & X_4 \arrow[ul, "\{b,m\}"'] \arrow[dl, "\{b,d,f\}"] \\
 & X_2 & & & X_3 \arrow[lll, "\{u\}"] & \\
 & & X_1 \arrow[ul, "\{m\}"] \arrow[urr, "\{m\}"'] & & &
\end{tikzcd}$$

X_1 :	*forecast*
X_2 :	*rain*
X_3 :	*dry*
X_4 :	*mowing*
X_5 :	*watering*
X_6 :	*dishwashing*
X_7 :	*ventilating*
X_8 :	*washing machine*
X_9 :	*warming*

Figure 4.5: Smart home example.

Verification of Parallel Programs

Some techniques for program verification make use of $\mathcal{IA}$ (see, e.g. [179]). Verifying parallel programs is a challenging task [54] which may take advantage from a branching interval algebra such as $\mathcal{BA}$, in which the typical *fork* constructs can be modelled in a natural way. Consistency, in this case, can be interpreted as the absence of temporal contradictions in the executions of subroutines. One might think that $\mathcal{IA}$ already has the necessary relations to model a problem in which several processes execute on a parallel architecture.

One might think that $\mathcal{IA}$ already has the expressiveness to model a problem in which several processes run on a parallel architecture, but this is not the case. Let p_1 and p_2 be two concurrent processes, if p_1 *overlaps* p_2 it means that the two processes, at some point, are running at the same time. The same can be said for other relations such as: *during*, *starts*, *finishes*, *equals*. $\mathcal{IA}$ relations are excellent for modelling processes that at some point must execute simultaneously, perhaps because they must communicate with each other. In $\mathcal{IA}$, on the other hand, we have no possibility of modelling two or more processes that could execute at the same time but have no constraints between them. Processes of this type could also be executed on different processors. The latter case is instead easily modelled in $\mathcal{BA}$ with the *unrelated* relation.

4.3 Tractability in Qualitative Algebras: the State of the Art

In this section we will briefly recap the main tractability results in the context of the $\mathcal{PA}$, $\mathcal{IA}$, $\mathcal{BPA}$ and $\mathcal{BA}$.

4.3.1 Linear Point and Interval Algebras

In [192], the authors show that SAT($\mathcal{IA}$) is NP-complete, and introduce the concept of *convex* (or *continuous*) relations in the point algebra. A relation R is *convex* if, given the instance $\Theta = \{XRY\}$ and two models $\hat{\Theta}_1, \hat{\Theta}_2$ such that $\hat{\Theta}_1(X) < \hat{\Theta}_2(X)$, it is the case that all the interpretations $\hat{\Theta}_i$ such that $\hat{\Theta}_1(X) < \hat{\Theta}_i(X) < \hat{\Theta}_2(X)$ are also models for Θ. For example, the $\mathcal{PA}$ relations $\{>\}$ and $\{\leq\}$ are convex, while $\{<,>\}$ is not.

By leveraging the properties of such relations, it is possible to prove that the *Path Consistency* algorithm decides MIN($\mathcal{PA}_{convex}$), where $\mathcal{PA}_{convex}$ is the weak subalgebra of the $\mathcal{PA}$ that only contains convex relations. Since deciding MIN implies deciding SAT, the tractability (and PC-tractability) of $\mathcal{PA}_{convex}$ follows. This result can be immediately extended to the convex fragment of the $\mathcal{IA}$, that is, $\mathcal{IA}_{convex}$, which is the weak subalgebra of the $\mathcal{IA}$ that only contains relations whose point-mappings are relations in $\mathcal{PA}_{convex}$. In [191], this analysis was extended to the whole $\mathcal{PA}$, and it was shown that SAT is still PC-tractable for it, which, in turn, entails the PC-tractability of the consistency problem of $\mathcal{IA}_{point}$, which is clearly an extension of $\mathcal{IA}_{convex}$. However, MIN($\mathcal{PA}$) is no longer PC-tractable (it is, however, decided by a, still polynomial, generalization of the *Path Consistency* algorithm).

In [154], Nebel and Bürckert define an ad-hoc Horn ontology called *ORD*, together with a particular kind of clauses, called ORD, and show that every $\mathcal{IA}$ relation can be represented by a set of such clauses.

An ORD clause only contains conjunctions of literals of the form:

$$x = y, x \leq y, x \neq y$$

it does not contain negations of atoms of the form $x \leq y$ (denoted by $\nleq$). Note that $x < y$ can be written as $x \leq y \wedge x \neq y$.

Definition 4.3.1 (*ORD* theory)**.** *ORD* is an order theory with the following axioms:

- $\forall x : (x \leq x)$ (reflexivity)
- $\forall x, y : (x \leq y) \wedge (y \leq x) \Rightarrow (x = y)$ (anti-symmetry)
- $\forall x, y, z : (x \leq y) \wedge (y \leq z) \Rightarrow (x \leq z)$ (transitivity)
- $\forall x, y : (x = y) \Rightarrow (x \leq y)$ (left weakening)
- $\forall x, y : (x = y) \Rightarrow (y \leq x)$ (right weakening)

Then, they defined $\mathcal{IA}_{Horn}$ as the set of relations which can be represented using only ORD clauses that also in Horn form, called ORD-HORN clauses. A clause (i.e., a disjunction of literals) is called a Horn clause if it contains at most one positive literal. $\mathcal{IA}_{Horn}$ is also PC-tractable, since *Path Consistency* in $\mathcal{IA}$ is shown to be equivalent to Positive Unit Resolution (PUR), which is refutation complete for Horn theories [103].

Definition 4.3.2 (Positive Unit Resolution PUR)**.** *Resolution* is a logical inference rule that from two clauses containing complementary literals produces a new clause containing all the literals of the two original clauses except the two complements with the following scheme:

$$(p_1 \vee \cdots \vee p_n \vee q) \wedge (r_1 \vee \cdots \vee r_m \vee \neg q) \models (p_1 \vee \cdots \vee p_n \vee r_1 \vee \cdots \vee r_m)$$

Unit resolution restricts one of the two involved clauses to be a unit clause (a clause composed of only one literal).

Positive Unit Resolution PUR, requires for the unit clause to be positive (i.e., a positive literal).

$\mathcal{IA}_{Horn}$, which extends $\mathcal{IA}_{point}$, is also a maximal tractable subalgebra, and the only one that includes $\mathcal{IA}_{basic}$.

While for both $\mathcal{IA}_{Horn}$ and $\mathcal{IA}_{point}$ one needs stronger algorithms to prove minimality, there are a couple of subsets of $\mathcal{IA}_{Horn}$ [35, 7], quite bigger than $\mathcal{IA}_{convex}$, for which *Path Consistency* is still sufficient to decide MIN. An alternative proof for the tractability of the $\mathcal{IA}_{Horn}$ fragment was given by Ligozat [134] while studying k-interval algebras over total orders (${}^k\mathcal{IA}$) [133] (in ${}^k\mathcal{IA}$, intervals have k endpoints, so $\mathcal{PA} = {}^1\mathcal{IA}$ and $\mathcal{IA} = {}^2\mathcal{IA}$). A further step in delineating the border of tractability in the $\mathcal{IA}$ was done in [62], where the authors introduce the concept of *acyclic* relations, i.e. relations that can never be satisfied in a cycle, like $\{b\}, \{m\}, \{bi, oi\}, \ldots$. Building on this notion, they find 21 PC-tractable subalgebras, nine of which, called $\mathcal{A}_1, \ldots, \mathcal{A}_4$, $\mathcal{B}_1, \ldots, \mathcal{B}_4$ and $\mathcal{IA}_{\equiv}$, are also maximal. In particular, the $\mathcal{IA}_{\equiv}$ is the set of relations that contains equality ($\mathcal{IA}_{\equiv} = \{R \mid \{e\} \subseteq R\}$). Clearly, this fragment is polynomially satisfiable by assigning $\{e\}$ to all relations. The remaining 12 subalgebras were soon extended and merged into eight so-called *starting-point* and *ending-point* algebras [63] (denoted $\mathcal{S}_1, \ldots, \mathcal{S}_3$, $\mathcal{E}_1, \ldots, \mathcal{E}_3$ and $\mathcal{S}^*, \mathcal{E}^*$), which are all maximally tractable, but not PC-tractable. All of the 18 maximal algebras mentioned so far were proven to be maximal by exhaustively generating all their possible extensions and showing that they always ended up containing one of the following small intractable fragments [63]:

$$
\begin{aligned}
\mathcal{N}_1 &= \{\{b, di, fi, m, o\}, \{b, d, m, o, s\}, \{d, di, fi, oi, si\}\}\\
\mathcal{N}_2 &= \{\{b, di, fi, m, o\}, \{b, d, m, o, s\}, \{di, fi, o, oi, si\}\}\\
\mathcal{N}_3 &= \{\{b, bi\}, \{o, oi\}\}\\
\mathcal{N}_4 &= \{\{b, bi\}, \{m, mi, o, oi\}\}\\
\mathcal{N}_5 &= \{\{m, mi\}, \{b, bi, f, fi, s, si\}\}\\
\mathcal{N}_6 &= \{\{b, bi, m, mi\}, \{o, oi\}\}
\end{aligned}
$$

As we have observed in the previous section, the above classification is only exhaustive for weak subalgebras. One example of the non-triviality of this question is the fragment Δ_1, described in [94], which is not contained in any of the above fragments and, yet, whose normalized consistency is tractable.

4.3.2 Branching Point and Interval Algebras

The tractability of the $\mathcal{BPA}$ was already proven by Hirsh [105] as part of a broader study of algebraic logic, but a further step is done in [44], where Broxvall studies the computational properties of the disjunctive $\mathcal{BPA}$, which extends the standard $\mathcal{BPA}$ by allowing an arbitrary number of disjunctions between constraints. Formally, given a (conjunctive) algebra $\mathcal{A}$ and two fragments $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathcal{A}$, we can build a disjunctive set $\mathcal{S}_1 \overset{\times}{\vee} \mathcal{S}_2$ which contains the binary disjunctions of the relations in $\mathcal{S}_1$ and $\mathcal{S}_2$. A fragment can be combined with itself, and we have that $\mathcal{S}^1 = \mathcal{S}$ while $\mathcal{S}^i = \mathcal{S}^{i-1} \overset{\times}{\vee} \mathcal{S}$. In particular, the set $\mathcal{S}^* = \bigcup_{i=0}^{\infty} \mathcal{S}^i$, allows an arbitrary number of disjunctions in its relations. Disjunctive fragments of point algebras are interesting for us because we can use them to represent the point-mapping of the corresponding interval fragments. In the linear setting, for example, $\mathcal{IA}_{Horn}$ can be translated to a certain disjunctive fragment of $\mathcal{PA}^*$ (although, to our knowledge, this relationship has not been studied in the linear case); in fact it is intuitive to see that any relation in ${}^k\mathcal{IA}$, for some fixed k, can be translated into an equivalent set of relations over $\mathcal{PA}^*$, while the converse is not true in general, and therefore, for a fixed topology, disjunctive point algebras offer more expressive power than k-interval algebras. There are exactly five maximal tractable subalgebras of $\mathcal{BPA}^*$, which can be obtained by combining the sets defined in Tab. 4.3:

$$
\begin{aligned}
&\mathcal{T}_A = \Gamma_A && \mathcal{T}_B = \Gamma_B \overset{\times}{\vee} \Delta_B^* && \mathcal{T}_C = \Delta_C^*\\
&\mathcal{T}_D = \Gamma_D \overset{\times}{\vee} \Delta_D^* && \mathcal{T}_E = \Gamma_E \overset{\times}{\vee} \Delta_E^*. &&
\end{aligned}
$$

For what concerns the $\mathcal{BA}$, on the other hand, until recently, the only fragment that was known to be tractable (and in particular PC-tractable) was $\mathcal{BA}_{basic}$ [168]. Moreover, the authors also state that $\mathcal{BA}_{point}$ is intractable, but one can already see that this cannot be the case.

	Γ_A	Γ_B	Δ_B	Δ_C	Γ_D	Δ_D	Γ_E	Δ_E
$<$	✓	✓					✓	
$\leq$	✓	✓		✓			✓	
$\lessgtr$	✓	✓	✓				✓	
$\lesseqgtr$	✓	✓	✓	✓			✓	
$\|$	✓				✓	✓	✓	
$=\|$	✓			✓	✓	✓		
$=$	✓	✓		✓	✓		✓	
$\neq$	✓	✓	✓		✓	✓	✓	✓
$<\|$	✓				✓	✓	✓	
$\leq\|$	✓			✓	✓	✓		
$\top$	✓	✓	✓	✓	✓	✓	✓	✓

Table 4.3: Broxvall's basic tractable fragments. For compactness, converse relations are not shown but are implied to be included whenever the direct relation is included.

4.4 Tractability in the Branching Interval Algebra

In this section we discuss our findings regarding the tractability and the PC-tractability of fragments of the $\mathcal{BA}$.

4.4.1 The Convex Fragment

As recalled in the previous sections, the convex fragment of the $\mathcal{PA}$ can be easily enumerated:

$$\mathcal{PA}_{convex} = \{\{=\}, \{<\}, \leq, \geq, \{>\}, \top\}.$$

It can be easily checked that $\mathcal{PA}_{convex}$ is a weak subalgebra of the $\mathcal{PA}$, the only missing relation from $\mathcal{PA}$ being $\{<, >\}$. We can extend $\mathcal{PA}_{convex}$ to the branching case, and define the *convex branching point fragment* as the set:

$$\mathcal{BPA}_{convex} = \{\{=\}, \{<\}, \leq, \geq, \{>\}, \{\|\}, \top_{\mathcal{PA}}\}.$$

The set $\mathcal{BPA}_{convex}$ is closed under under converse, intersection and strong composition, but, unfortunately, not under weak composition; for example, $\{\|\} \diamond \{\|\} = \top$. Thus, $\mathcal{BPA}_{convex}$ is a strong subalgebra of the $\mathcal{BPA}$. $\mathcal{BPA}_{convex}$ is naturally associated with its branching interval version:

$$\mathcal{BA}_{convex} = \{R \mid R \in \mathcal{BA} \wedge \xi(R) \in \mathcal{BPA}_{convex}\}$$

Clearly, $\mathcal{BA}_{convex}$ inherits from $\mathcal{BPA}_{convex}$ its closure properties, and it is therefore closed under strong, but not weak, composition. As we have recalled, in the linear setting, both in the point-based and the interval-based ontology, both minimality and consistency are solved by *Path Consistency*. As we shall see, this happens in the branching setting as well.

Theorem 4.4.1. *The Path Consistency algorithm decides both* $\mathrm{MIN}(\mathcal{PA}_{convex})$ *and* $\mathrm{MIN}(\mathcal{BPA}_{convex})$*, and therefore both* $\mathrm{SAT}(\mathcal{PA}_{convex})$ *and* $\mathrm{SAT}(\mathcal{BPA}_{convex})$*. As a consequence,* $\mathcal{BPA}_{convex}$ *is tractable.*

Proof. Let Θ be an $\mathcal{BA}_{convex}$-instance and let Θ' be the $\mathcal{BPA}_{convex}$-instance that results from translating Θ into the language of endpoints. We assume that *Path Consistency* has been forced on Θ, and we want to show that Θ is strongly k-consistent for every k; since a strongly k-consistent instance must have minimal labels, we have the result. Let us proceed by induction. As base case, we know that Θ is k-consistent for $k \leq 3$. As for the inductive case, we suppose now that Θ is $k-1$-consistent and we prove that it is also k-consistent. Consider a subset $S = \{X_1, \ldots, X_{k-1}\}$ of $k-1$ interval variables in Θ. Let us call Θ_{k-1} the projection of Θ restricted to the variables in S and the constraints among them, and let us call Θ'_{k-1} the corresponding $\mathcal{BPA}_{convex}$-instance, whose variables are precisely the $2 \cdot (k-1)$ endpoints of $X_1, \ldots, X_{k-1}$. Our strategy can be summarized as follows: since N_{k-1} is consistent by hypothesis, M_{k-1} must be consistent as well, that is, it must be realized in a branching model $\mathcal{T}_{k-1}$; if we pick the point variables corresponding to the endpoints of any k-th interval variable and accommodate them in a branching model $\mathcal{T}_k$ showing that every constraint is respected, then we obtain a model for k interval variables, proving that N_k is also consistent. Let X be any interval variable in Θ different from $X_1, \ldots, X_{k-1}$, and let XR_iX_i the $\mathcal{BA}_{convex}$-relation between the variables X and X_i, for each i. Let M_k be the $\mathcal{BPA}_{convex}$-network obtained by adding to Θ'_{k-1} the point variables X^-, X^+, the constraint $X^- < X^+$, and every constraint between the endpoints of X and the endpoints of $X_1, \ldots, X_{k-1}$ that results from translating the constraints of the type XR_iX_i. On $\mathcal{T}_{k-1}$ we can identify the set $\mathcal{R} = \{p_1, \ldots, p_n\}$ with the following characteristics: for each i, p_i is the realization of some point variable y in Θ'_{k-1} (that is, p_i is the realization of some endpoint of the interval variables $X_1, \ldots, X_{k-1}$) and that, for every point variable $y \in \Theta'_{k-1}$, realized in some point $p \in \mathcal{T}_{k-1}$, it is not the case that $p < p_i$. Indeed, consider the branching model $\mathcal{T}_{k-1}$: since it is a tree, it may be the case that, in order to realize two variables that are constrained to be incomparable to each other, a greatest common predecessor must be added; therefore, if projected to the points that realize some point variable in Θ_{k-1}, $\mathcal{T}_{k-1}$ is a *forest of trees*, rather than a tree. Every point in $\mathcal{R}$ is the *root* of one of the trees in $\mathcal{T}_{k-1}$; let us call q their greatest common predecessor. Now, let $x_1, \ldots, x_m$ be the point variables that have been realized in $p_1, \ldots p_n$ (observe that $n \leq m \leq k-1$: two variables may have been realized in the same root, and m cannot exceed the number of intervals $k-1$ because the rightmost endpoint of each interval cannot be a root). We want to show, first, that the point variable X^- can be successfully realized on $\mathcal{T}_{k-1}$, and we proceed case by case.

- Suppose, first, that $(x_l, X^-) = \{\|\}$ for every point variable x_l realized in some root. In this case, we realize X^- with a new point p such that $p \parallel p_i$ for each root p_i, and that $q < a$. To prove that this is a consistent choice, consider any point variable y of M_{k-1} realized at some point $s \geq p_i$ for some root p_i. Suppose that p_i is the realization of some point variable x_l, which means that $(y, x_l) \subseteq \top_{\mathcal{PA}}$. If $< \in (y, x_l)$, then $(y, x_l) \circ (x_l, X^-) = \{<, \|\}$. By intersection with $\mathcal{BA}_{convex}$, then either $(y, X^-) = \{<\}$ or $(y, X^-) = \{\|\}$; in the first case, however, we obtain, by path consistency, that $(X^-, x_l) \in \top_{\mathcal{PA}}$, which is a contradiction. Therefore, $(y, X^-) = \{\|\}$. If, on the other hand, $< \notin (y, x_l)$, then, $(y, x_l) \subseteq \{>, =\}$, and, since $\{>, =\} \circ \{\|\} = \{\|\}$, it must be the case that $(y, X^-) = \{\|\}$.

- Suppose, now, that $(x_l, X^-) \subseteq \top_{\mathcal{PA}}$ for some point variable x_l realized in some root p_i. Observe, first, that if $(X^-, x_l) = \{<\}$, then we can select the subset $\mathcal{R}' \subseteq \mathcal{R}$ such that, for each $x_l' \in \mathcal{R}'$, we have $(X^-, x_l') = \{<\}$; in this case, by the argument in the above case, for each $x_l'' \in \mathcal{R} \setminus \mathcal{R}'$, we have $(X^-, x_l') = \{\|\}$. Consider each p_j that is the realization of some variable in $\mathcal{R}'$: we realize X^- in a point $p > q$, such that p is less than every such p_j, and incomparable with every other root in $\mathcal{R} \setminus \mathcal{R}'$. If, otherwise, $(X^-, x_l) \subseteq \{>, =\}$, then, for each x_l' realized in some root $p_j \neq p_i$, we must have $(X^-, x_l') = \{\|\}$. In this case, we can say that p_i is the root of the tree in which we have to realize X^-; let us call it $\mathcal{T}_{p_i}$. Observe that, by the same argument as in the above case, wherever we realize X^- in $\mathcal{T}_{p_i}$, this realization is consistent with any point that belongs to some $\mathcal{T}_{p_j}$ with $p_j \neq p_i$. Now, we consider the point $s \in \mathcal{T}_{p_i}$ which is the least point (greater than or equal to p_i) with at least two immediate successors s_1, s_2 such that $s_1 \parallel s_2$, if it exists. We have the following cases.

  - Suppose that s does not exist. This means that T_{p_i} is linearly ordered. Let s' be the least point (greater than p_i), such that is the realization of some variable y such that $(y, X^-) = \{\|\}$. If there is no such s', then, by Theorem 4.4.1, we can find a realization for X^- consistent with $\mathcal{T}_{p_i}$; since we already know that such a realization is consistent with every other tree, we conclude that it is consistent. If s' exists, then we realize X^- in a point p such that $p \parallel s'$ and that $p > t$ where t is the immediate predecessor of s'. By the argument used in the first case, this choice must be consistent with $\mathcal{T}_{p_i}$, and therefore it must be consistent.

– Suppose, now, that b exists. If y is realized in b, and $\{<, =, \|\} \cap (X^-, y) \neq \emptyset$, then we proceed as in the previous case. f, on the other hand, $(X^-, y) = \{>\}$, we have the following two cases. First, if $< \in (X^-, y)$ for every y_j realized in some point s_j such that s_j is immediate successor of s, then realize X^- in a point p such that $s < p$ and that $p < s_j$ for every immediate successor s_j of s, which must be a consistent choice, given that, by path consistency, the relation between X^- and every variable realized in a point greater than s must contain $<$. If, for some immediate successor s_j of s, which is the realization of some variable y, it is the case that $< \notin (X^-, y)$, then we can treat every immediate successor s_j of s as the root of some sub-tree of T_{p_i}, and therefore we can apply the same entire argument, recursively.

Having realized the variable X^-, the network Θ'_{k-1} enriched with X^- (and all relative constraints) must be consistent. By reapplying the entire argument, we can show that any other point variable can be consistently realized in the resulting network; if we choose X^+ among these, we prove that the original network Θ is, in fact k-consistent, completing the induction. □

The set $\mathcal{BPA}_{convex}$ is not maximal with respect to (PC-)tractability. It is, however, maximal with respect to *Path Consistency* being able to decide MIN for it. Let $R \neq \emptyset$ be any $\mathcal{BPA}$-relation not included in $\mathcal{BPA}_{convex}$. We say that $\mathcal{BPA}_{convex} \cup \{R\}$ is *strongly non-convex* if there exists a path consistent $\mathcal{BPA}_{convex} \cup \{R\}$-network with non-minimal labels, and that it is *weakly non-convex* if there exists a $\mathcal{BPA}_{convex} \cup \{R\}$-network in which minimal labels cannot be forced by path consistency. The difference between the above two definitions is subtle. In the first case, the counterexample is a network N over which path consistency has been forced, but some labels are not minimal. In the second case, it is a network N over which path consistency has not been forced yet, and such that forcing its path consistency results in a new network N', which is path consistent but with some non-minimal label. Observe, now, that proving that a certain extension of $\mathcal{BPA}_{convex}$ is weakly non-convex is sufficient to prove that our method cannot be applied on it. Thus, we say that $\mathcal{BPA}_{convex}$ is *weakly maximal.*

Theorem 4.4.2. *Let $R \neq \emptyset$ be any $\mathcal{BPA}$-relation not included in $\mathcal{BPA}_{convex}$. Then $\mathcal{BPA}_{convex} \cup \{R\}$ is weakly non-convex.*

Proof. We have proved this result in a computer-assisted way. For each $R \in \mathcal{BPA}$ such that $R \neq \emptyset$, we have systematically generated all 4-node networks, searching for a witness of strong non-convexity. After such a systematic exploration, we have found three witnesses of strong non-convexity, namely for the relations $\{<, >\}$, $\{<, \|\}$, and $\{=, \|\}$; no witness for the relation $\{>, \|\}$ is needed, as it is the inverse of a relation for which a witness is already known. Then, for the remaining cases, we have adapted the already found witnesses to become counterexamples for the weak non-convexity of the remaining extensions of $\mathcal{BPA}_{convex}$, completing the proof.

Consider, first, N_1 (Fig. 4.6), which contains constraints from the set $\mathcal{BPA}_{convex} \cup \{<, >\}$. Clearly, N_1 is path consistent; but, as it can be easily checked, its labels are not minimal, and, in particular, the constraint $x_1 = x_3$ cannot be realized. As a matter of fact, if we instantiate x_1, x_2, x_3 and x_4 with the points p, q, s and t, respectively, then we observe immediately that they cannot be all equal (because of the constraint between x_2 and x_4). So, if $p = q$, then either $q < p, s$ or $t > p, q$. In the first case, the constraint $x_1\{>, =\}x_2$ cannot be satisfied; in the second case, $x_1\{>, =\}x_2$ cannot be satisfied. So, N_1 has non-minimal labels.

Now, consider the network N_2 (Figure 4.6), which contains constraints from the set $\mathcal{BPA}_{convex} \cup \{<, \|\}$ and, as before, is path consistent. The only consistent realization for x_1, x_2, x_3, x_4 is with points p, q, s, t, respectively, on a linear model; otherwise, the constraints that start at x_1 could not be satisfied. But the assignment $p = q$ cannot be extended to any assignment that satisfies all constraints: in fact, we must have $q > s$, $s > t$, and $t > q$, which is impossible. So, N_2 does not have minimal labels.

Finally, consider the network N_3 (Figure 4.6), which contains constraints from the set $\mathcal{BPA}_{convex} \cup \{=, \|\}$, and is path consistent. In this case, again, every consistent assignment of the variables x_1, x_2, x_3, x_4 to the points p, q, s, t is such that all points lay on the same linear model, which makes the constraint $x_1\{=, \|\}x_2$ non minimal, as it cannot be that $p \parallel q$.

Now, we prove that $\mathcal{BPA}_{convex} \cup \{<, >, \|\}$ is weakly non-convex by observing the network N_4 (Figure 4.7). Applying *Path Consistency* to N_4 implies that $x_2\{<, >, \|\}x_4$ becomes $x_2\{<, >\}x_4$. The resulting network is in fact symmetric to the network N_1, for which we have already shown that it does not have minimal labels. So, N_4 is a witness for the weak non-convexity of $\mathcal{BPA}_{convex} \cup \{<, >, \|\}$. Similarly, $\mathcal{BPA}_{convex} \cup \{=, <, \|\}$ is weakly non-convex as witnessed by the network N_5 (Figure 4.8), once *Path Consistency* has been applied to it. As a matter of fact, by assigning x_1, x_2, x_3, x_4 to p, q, s, t, respectively, $p < t$ cannot be satisfied: if $q = p$, then the constraint $x_2\{=, >, \|\}x_4$ is violated, and if $q \parallel p, t$, then the pair of constraints $x_2\{=, >\}x_3$ and $x_3\{=, <\}x_4$ cannot be satisfied together.

Finally, as for the weak non-convexity of $\mathcal{BPA}_{convex} \cup \{\top\}$, consider the network N_6 in Figure 4.9. Applying *Path Consistency* to N_6 leads to a network that, projected to the nodes x_1, x_2, x_3, x_4, is precisely N_2, which, as we have already proved, does not have minimal labels. □

A natural question, at this point, is if the convex fragment of the $\mathcal{BA}$ can be extended preserving, at least, its tractability. Following the path traced for the linear case, we introduce now the Horn fragment.

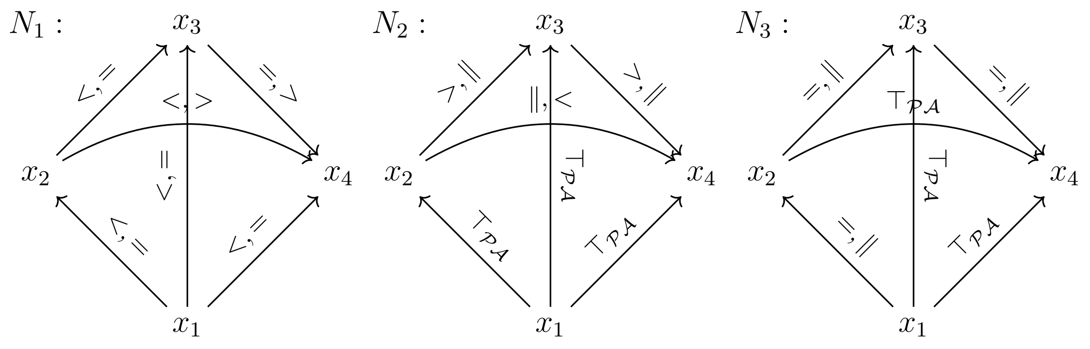


Figure 4.6: Witnesses for the strong non-convexity of the extensions of $\mathcal{BPA}_{convex}$ with, respectively $\{<,>\}$ (N_1), $\{<,\|\}$ (N_2), and $\{=,\|\}$ (N_3).

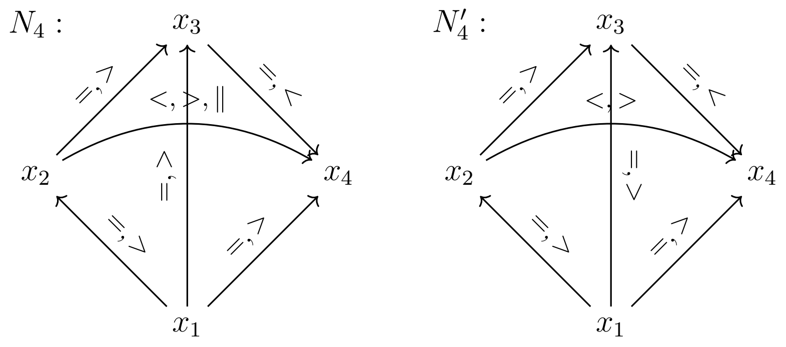


Figure 4.7: A witness for the weak non-convexity of the extensions of $\mathcal{BPA}_{convex}$ with $\{<,>,\|\}$ (N_4) and the result N_4' of enforcing path consistency in N_4.

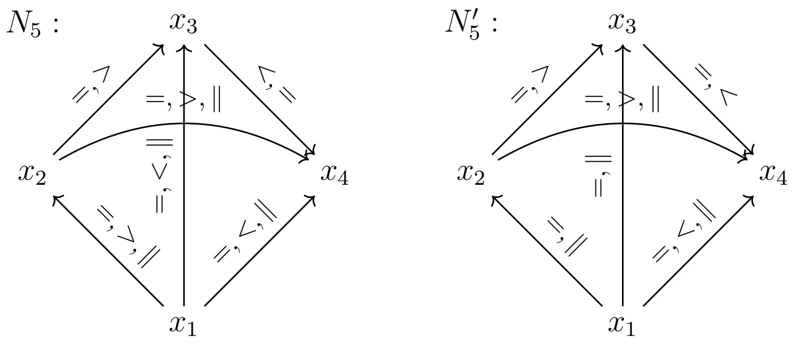


Figure 4.8: A witness for the weak non-convexity of the extensions of $\mathcal{BPA}_{convex}$ with $\{=,<,\|\}$ (N_5) and the result N_5' of enforcing path consistency in N_5.

$$\begin{tikzcd}
N_6 : & x_{2,3} \arrow[rr, ">"] & & x_3 \arrow[r, "\parallel"] & x_{3,4} \arrow[dd, "\vee"] \\
& & x_{2,4} \arrow[drr, bend left, "/\!\!/"] & & \\
& x_2 \arrow[uu, "="] \arrow[ur, "<"] & & & x_4 \\
& & & x_1 \arrow[ull, "\top_{\mathcal{PA}}"] \arrow[uuu, "\top_{\mathcal{PA}}"] \arrow[ur, "\top_{\mathcal{PA}}"] &
\end{tikzcd}$$

Figure 4.9: A witness for the weak non-convexity of the extensions of $\mathcal{BPA}_{convex}$ with $\{\top\}$ (missing edges are precisely those with the relation $\top$).

4.4.2 The Horn Fragment

In any given temporal algebra $\mathcal{A}$, constraints involving basic relations can be seen as *literals* over the language of the domain of $\mathcal{A}$; constraints involving relations are *clauses*, and instances are *formulae* in Conjunctive Normal Form (CNF). For example, deciding the consistency of the instance:

$$\Theta = \{X\{bi, e\}Y,\ X\{m, f\}Z,\ Y\{di\}Z\}$$

is equivalent to deciding the consistency of the formula:

$$\Phi = (XbiY \vee XeY) \wedge (XmZ \vee XfZ) \wedge (YdiZ) \wedge \mathcal{A}(\Theta)$$

where $\mathcal{A}(\Theta)$ denotes the axioms of $\mathcal{A}$ instantiated to the variables in Θ. In this sense, point-mapping provides a mean to translate any *interval-literal* to a *point-clause*, and any *interval-clause* to a Disjunctive Normal Form (DNF) point-formula, which can then be reformulated to an equivalent *minimal* CNF point-formula[1]. Let us also denote with $C, D, \ldots$ (resp., $\Phi, \Psi, \ldots$) a generic clause (resp., set of clauses). This approach leads to the identification, in the linear case, of the fragment $\mathcal{IA}_{Horn}$ of the $\mathcal{IA}$, which is not only (PC-)tractable, but also maximally so. We now extend it to the branching setting. In particular, we shall devise an appropriate first-order Horn *pre-theory* of tree orders, called *TORD*, together with a language of *allowed literals*, called TORD. We shall then show that a formula in the language TORD over the theory *TORD* can be modelled on a tree order if and only if it is satisfiable. Finally, by defining a mapping operator π that maps every $\mathcal{BA}$ relation, and in turn any instance, to a (particular) equivalent set of TORD clauses, and selecting those that are Horn, we shall obtain a fragment whose consistency problem can be decided in deterministic polynomial time.

[1]The sets of basic/non-basic/point/interval relations are fixed, their translations are constant in size, and can be computed in constant time by using a lookup table. Such constants are generally in the order of the cardinality of the interval algebra.

Definition 4.4.1. The TORD language encompasses an enumerable set of variables $X, Y, \ldots$ and the binary relations $\doteq$ (*equality*), $\preceq$ (*less or equal*), $\sim$ (*linear*), $\shortparallel$ (*incomparable*), and $\prec\shortparallel$ (*less or incomparable*).

In this context, the theory of future branching models of time cannot be (fully) axiomatized in the standard way, because some of the necessary properties cannot be put in form of Horn formulas (e.g., $X \sim Y$ as $X \preceq Y \vee Y \preceq X$). To our purposes, however, it suffices to have models that can be *extended to* tree-like orderings.

Definition 4.4.2. The *TORD theory* is characterized by the following axioms:

$$\begin{array}{llll}
\forall x: & (x \doteq x) & \text{(reflexivity)} & (4.1) \\
\forall x, y: & (x \doteq y) \Rightarrow (x \preceq y) \wedge (y \preceq x) & \text{(weakening)} & (4.2) \\
\forall x, y: & (x \preceq y) \wedge (y \preceq x) \Rightarrow (x \doteq y) & \text{(antisymmetry)} & (4.3) \\
\forall x, y, z: & (x \preceq y) \wedge (y \preceq z) \Rightarrow (x \preceq z) & \text{(transitivity)} & (4.4) \\
\forall x, y: & (x \preceq y) \Rightarrow (x \sim y) \wedge (y \sim x) & \text{(weakening)} & (4.5) \\
\forall x, y: & (x \sim y) \Rightarrow (y \sim x) & \text{(symmetry)} & (4.6) \\
\forall x, y: & (x \sim y) \Rightarrow (x \nshortparallel y) & \text{(incompatibility)} & (4.7) \\
\forall x, y: & (x \shortparallel y) \Rightarrow (x \prec\shortparallel y) \wedge (y \prec\shortparallel x) & \text{(weakening)} & (4.8) \\
\forall x, y: & (x \prec\shortparallel y) \wedge (y \prec\shortparallel x) \Rightarrow (x \shortparallel y) & \text{(pseudo-antisymmetry)} & (4.9) \\
\forall x, y: & (x \prec\shortparallel y) \Rightarrow (y \npreceq x) & \text{(incompatibility)} & (4.10) \\
\forall x, y, z: & (x \shortparallel y) \wedge (y \preceq z) \Rightarrow (x \shortparallel z) & \text{(tree-likeness)} & (4.11)
\end{array}$$

In the following, we denote by *TORD* the set of axioms 4.1-4.11; observe that *TORD* is indeed a Horn theory. Moreover, as it can be easily shown, it is minimal. We use TORD to translate the relations of $\mathcal{BA}$; such a translation is correct if and only if the resulting model can be interpreted as a future branching model of time. It turns out that, in order to guarantee that this is possible, we need to further limit the use of TORD in translations, and, in particular, we say that C is an *admissible clause* if it uses only literals with the positive relations $\doteq$, $\preceq$, $\sim$, $\shortparallel$, $\prec\shortparallel$, and the negative relation $\not\doteq$. Observe that limiting the use of certain relations does not decrease the (semantic) expressive power of the language, because: $X \npreceq Y$ can be written as $Y \prec\shortparallel X$, $X \nsim Y$ can be written as $X \shortparallel Y$, $X \nshortparallel Y$ can be written as $X \sim Y$ and $X \not\prec\shortparallel Y$ can be written as $Y \preceq X$.

Admissible TORD clauses which are also Horn are denoted as TORD-Horn clauses.

Theorem 4.4.3. *Every model* $(\mathcal{M}, \doteq, \preceq, \sim, \shortparallel, \prec\shortparallel)$ *of* $TORD \cup \mathcal{C}$*, where* $\mathcal{C}$ *is a set of admissible clauses, can be represented as a branching model of time*[2]*.*

[2]Such a translation implicitly assumes that point variables $(I^-, I^+, \ldots)$ are correctly related by the relation $\{<\}$ in the object language.

Proof. Since $\doteq$ is an equivalence relation, we can take the quotient $\mathcal{M}/_{\doteq}$, denoted $\mathcal{T}$, and equipped with the canonical equivalence $=$. In the following, we denote by $x, y, \ldots$, rather than $[X]_{/\doteq}, [Y]_{/\doteq}$, the elements of $\mathcal{T}$. We define the binary relation $\leq$ between classes:

$$x \leq y \Longleftrightarrow^{def} \exists X \in x, Y \in y \mid X \preceq Y$$

and, consequently, $x < y$ as $x \leq y \wedge x \neq y$. We want to prove that $(\mathcal{T}, <)$ can be extended to a branching model of time.

- $\leq$ is an ordering relation. Clearly, $\leq$ is reflexive and antisymmetric because so is $\preceq$. Moreover, assume that $x \leq y$ and $y \leq z$ for some x, y, z. This means that $X \preceq Y$ and $Y' \preceq Z$ for some X, Y, Y', Z such that $X \in x$, $Y, Y' \in y$, and $Z \in z$. But since $Y, Y' \in y$, we have that $Y \doteq Y'$, and by axiom 4.2 we know that $Y \preceq Y'$. Since $\preceq$ is transitive, we obtain that $X \preceq Z$, implying that $x \leq z$. So, $\leq$ is also transitive. This also implies that $<$ is a strict pre-order, as it is irreflexive.

- $\leq$ can be extended to a tree-like order. To see this, observe that tree-likeness could be violated by having $x \nleq y$, $y \nleq x$, $y \leq z$, and $x \leq z$ for some x, y, z, but $\nleq$ simply cannot be generated by the set $\mathcal{C}$, since it contains only admissible clauses. Because we need to interpret every symbol of the language of TORD, let us define the *incomparable to* relation between classes:

  $$x \parallel y \Longleftrightarrow^{def} \exists X \in x, Y \in y \mid X \shortparallel Y$$

  which is irreflexive and symmetric because so is $\shortparallel$, so it is well-defined. To ensure that $(\mathcal{T}, <)$ can be extended to a tree-like ordering, we also have to guarantee that the introduction of $\parallel$ does not generate any contradictions. So, suppose that $x \parallel y, x \leq z$, and $y \leq t$ for some x, y, z, t. By definition, for some $X \in x$ and $Y \in y$ we have that $X \shortparallel Y$. Moreover, since $x \leq z$, for some $X' \in x$ and $Z \in z$ we have that $X' \preceq Z$. But this implies, by axiom 4.2, that $X \preceq Z$. So, axiom 4.11 applies, implying that $Y \shortparallel Z$. The same argument can be re-applied, leading us the conclusion that $Z \shortparallel T$. By definition, this implies that $z \parallel t$. By contradiction, assume now that $x \parallel y$ and $x \leq y$ for some x, y. This means that $X \shortparallel Y$ and $X' \preceq Y'$ for some $X, X' \in x$ and $Y, Y' \in y$. By axioms 4.2, 4.4 and 4.5, this implies that $X \sim Y$, which is in contradiction with axiom 4.7. As a consequence of these two facts we have that $x \parallel y \Leftrightarrow (x \nleq y \wedge y \nleq x)$ is realizable in $(\mathcal{T}, <)$. Now, let us define the *linear to* relation between classes:

  $$x \top_{\mathcal{PA}} y \Longleftrightarrow^{def} \exists X \in x, Y \in y \mid X \sim Y$$

  Clearly $\top_{\mathcal{PA}}$ is reflexive and symmetric because so is $\sim$, so it is well-defined. Once again, we need to make sure that introducing $\top_{\mathcal{PA}}$ does not generate contradictions. So, suppose, by contradiction, that $x \top_{\mathcal{PA}} y$ and $x \parallel y$ hold for some x, y. This means that $X \sim Y$ and $X' \shortparallel Y'$ for some $X, X' \in x$ and $Y, Y' \in y$. By axiom 4.2, this implies that $X \shortparallel Y$, which is in contradiction with axiom 4.7. Similarly, assume that $x \leq y$ for some x, y (the case in which $y \leq x$ or $x = y$ are similar). This means that $X \preceq Y$ for some $X \in x$ and $Y \in y$. By axiom 4.5, this implies that $X \sim Y$, leading us to conclude that $x \top_{\mathcal{PA}} y$, and, since $\mathcal{C}$ is admissible, $x \; \top_{\not{\mathcal{P}}\mathcal{A}} \; y \wedge x \nparallel y$ cannot occur. As a consequence, we have that $x \top_{\mathcal{PA}} y \Leftrightarrow (x \leq y \vee y \leq x \vee x = y)$ is realizable in $(\mathcal{T}, <)$. Finally, we define the *less than or incomparable to* relation between classes:

$$x <\!\parallel y \overset{def}{\Longleftrightarrow} \exists X \in x, Y \in y \mid X \prec\!\shortparallel Y$$

which is irreflexive and pseudo-antisymmetric because so is $\prec\!\shortparallel$, so it is well-defined. Suppose that, for some x, y it is the case that $x <\!\parallel y$ and $y \leq x$. This means that $X \prec\!\shortparallel Y$ and $Y' \preceq X'$ for some $X, X' \in x$ and $Y, Y' \in y$. But since $X, X' \in x$ and $Y, Y' \in y$, we have that $X \doteq X'$ and $Y \doteq Y'$, and by axioms 4.2, 4.4 and 4.5, we know that $X \sim Y$, which is in contradiction with axiom 4.7. Moreover, since $\mathcal{C}$ is a set of admissible clauses, $x \not<\!\parallel y \wedge y \not\leq x$ cannot occur. As a consequence, we have that $x <\!\parallel y \Leftrightarrow x < y \vee x \parallel y$ is realizable in $(\mathcal{T}, <)$.

In conclusion, the structure $(\mathcal{T}, <)$ can be extended to a branching model of time, as we wanted. □

It is important to remark that the use of an extended signature to specify the properties of a tree-like model is justified by the need of such a specification to be Horn. TORD-Horn clauses are expressive enough to translate a subset of $\mathcal{BA}$-relations that form an algebra, and allowing any of the forbidden symbols would require some non-Horn axiom to ensure isomorphism. The Horn fragment of the $\mathcal{BA}$ is defined as the following set:

$$\mathcal{BA}_{Horn} = \{R \mid R \in \mathcal{BA} \wedge \pi(R) \in \text{TORD-Horn}\}$$

Theorem 4.4.4. *$\mathcal{BA}_{Horn}$ is closed under converse, intersection, and weak composition, therefore it is a weak subalgebra.*

Proof. The proof was made by computer-assisted enumeration. The set can be considered closed under a certain operation if however one takes the elements of the set $\mathcal{BA}_{Horn}$ and performs the operation (according to the definitions introduced in Section 4.1) the result of the operation still belongs to the set $\mathcal{BA}_{Horn}$. □

The set $\mathcal{BA}_{Horn}$ consists of 4510 relations, and it extends the $\mathcal{BA}_{convex}$ fragment. Although it covers less than 1% of the entire algebra, it is about 50 times bigger than $\mathcal{BA}_{convex}$. Now, we can discuss its (PC-)tractability. Let us consider an

$(I^- \dot{\neq} J^-) \vee (I^+ \sim J^+)$	$(I^- \dot{\neq} J^-) \vee (I^+ \shortparallel J^+)$
$(I^+ \dot{\neq} J^+) \vee (I^- \doteq J^-)$	$(I^- \dot{\neq} J^-) \vee (I^+ \doteq J^+)$
$(I^+ \dot{\neq} J^+) \vee (I^- \preceq J^-)$	$(I^- \dot{\neq} J^-) \vee (I^+ \preceq J^+)$
$(I^+ \dot{\neq} J^+) \vee (J^- \preceq I^-)$	$(I^- \dot{\neq} J^-) \vee (J^+ \preceq I^+)$
$(I^+ \dot{\neq} J^+) \vee (I^- \prec\!\shortparallel J^-)$	$(I^- \dot{\neq} J^-) \vee (I^+ \prec\!\shortparallel J^+)$
$(I^+ \dot{\neq} J^+) \vee (J^- \prec\!\shortparallel I^-)$	$(I^- \dot{\neq} J^-) \vee (J^+ \prec\!\shortparallel I^+)$
$(I^- \dot{\neq} J^-) \vee (I^+ \dot{\neq} J^+)$	

Table 4.4: Enumeration of all the semantically different 2-literal TORD-Horn clauses.

instance Θ of $\mathcal{BA}_{Horn}$. By the above results, we know that Θ is consistent if and only if $\pi(\Theta) \wedge \mathit{TORD}(\Theta)$ is satisfiable. Now, we ask ourselves if the consistency of Θ can also be checked by path consistency. Again, following [154], proving that path consistency is complete for $\mathcal{BA}_{Horn}$ boils down to proving that, given the path consistent instance Θ, the empty clause cannot be derived from $\pi(\Theta) \wedge \mathit{TORD}(\Theta)$; to show the latter, one can restrict the attention to derivations that use *positive unit resolution*, which is complete for Horn clauses [103]. Let $\Theta = \{I_1 R_1 J_1, I_2 R_2 J_2, \ldots\}$ be a path consistent instance of $\mathcal{BA}_{Horn}$. Let $\pi(\Theta) = \{\varphi_1, \varphi_2, \ldots, \}$ be the TORD-Horn formulae resulting by translating the $\mathcal{BA}_{Horn}$-constraints of Θ. Interestingly, all the semantically different minimal TORD-Horn clauses are either unary or binary; in particular the literals in the binary ones enumerated in Tab. 4.4, always relate pairs of starting points or ending points, but never mixed pairs. In the following we assume that each formula φ_i is *explicit*, that is, it explicitly contains all consequences of every axiom of *TORD*, and that each clause $C_j \in \varphi_i$ is *minimal*, that is, it contains no redundant literal; a set $\pi(\Theta)$ in which every formula is explicit, and every clause in every formula is minimal will be called *explicit* and *clause-minimal*. We can now state the following theorem.

Theorem 4.4.5. *Let Θ be a path consistent instance of $\mathcal{BA}_{Horn}$. If $\pi(\Theta)$ is explicit and clause-minimal, the empty clause cannot be obtained from $\pi(\Theta) \wedge \mathit{TORD}(\Theta)$ by positive unit resolution.*

Proof. We prove a stronger claim, that is, we prove that if Θ is a path consistent instance of $\mathcal{BA}_{Horn}$, and $\pi(\Theta)$ is explicit and clause-minimal, then no new unit clause can be deduced by positive unit resolution from $\pi(\Theta) \wedge \mathit{TORD}(\Theta)$. As a matter of fact, to deduce a new unit clause, it must be the case that $\pi(\Theta) \wedge \mathit{TORD}(\Theta)$ contains one clause $C = \neg l_1 \vee \neg l_2 \vee \ldots \vee \neg l_q \vee l$, where $l_1, l_2, \ldots$ are propositional atoms, and a set of positive unit clauses $\mathcal{C} = \{C_1 = l_1, C_2 = l_2, \ldots, C_q = l_q\}$, but does not contain the clause $D = l$. Moreover, it must also be the case that $q \leq 2$, as we have observed that clauses of $\pi(\Theta)$ are at most binary, and instances of axioms are at most ternary. We proceed by case analysis.

- Suppose, first, that $C \in \pi(\Theta)$. If $\mathcal{C} \subseteq \pi(\Theta)$, its clauses must talk about the endpoints of different interval variables, otherwise no resolution step could be applied. Since any interval relation can be translated with clauses of the kind $C = (X \cdot Y \vee X \not\doteq Y)$ and $\mathcal{C} = \{(X \doteq Y)\}$ (since C is a Horn clause, $\cdot$ is positive). But, $\cdot \notin \{\doteq, \preceq, \sim\}$, otherwise $\pi(\Theta)$ would not be explicit, and $\cdot \notin \{\shortparallel, \prec\shortparallel\}$, otherwise $\pi(\Theta)$ would not be clause-minimal. Therefore, if $\mathcal{C} \subseteq \pi(\Theta)$ we cannot derive anything new. Clearly $\mathcal{C} \nsubseteq \mathit{TORD}(\Theta)$, since the only unitary axiom is the reflexivity axiom, from which we cannot derive new clauses.

- Suppose, then, that C is an instance of the transitivity axiom. As before, $\mathcal{C} \nsubseteq \mathit{TORD}(\Theta)$, so suppose that $\mathcal{C} \subseteq \pi(\Theta)$. To resolve a new clause from the transitivity axiom, it must be the case that $\mathcal{C} = \{(X \preceq Y), (Y \preceq Z)\}$. If the clauses in $\mathcal{C}$ talk about the endpoints of a single interval, then $(Y \preceq Z) \in \pi(\Theta)$ because of the explicitness assumption. So, it must be the case that C_1 belongs to the translation of some constraint $I(R_1)J$, and C_2 belongs to the translation of some constraint $J(R_2)K$. Since Θ is path consistent, the constraint $I(R_3)K$ exists, and $\pi(\Theta)$ contains its translation. Since (*i*) $R_3 \subseteq R_1 \diamond R_2$, (*ii*) weak composition computes the *strongest implied relation* given two intervals (or points), and (*iii*) $(X \preceq Z)$ would be resolved solely from C and $\mathcal{C}$, it must be the case that $(X \preceq Z) \in \pi(\Theta)$, so, also in this case, no new deduction can be performed.

- Assume, therefore, that C is an instance of the tree-likeness axiom, we apply a similar reasoning as before. Again, $\mathcal{C} \nsubseteq \mathit{TORD}(\Theta)$, so it must be a subset of $\pi(\Theta)$. To resolve a new clause from the tree-likeness axiom, it must be the case that $\mathcal{C} = \{(X \shortparallel Y), (Y \preceq Z)\}$. The clauses in $\mathcal{C}$ cannot talk about the endpoints of a single interval, because otherwise there would be an inconsistency. So, it must be the case that C_1 belongs to the translation of some constraint $I(R_1)J$, and C_2 belongs to the translation of some constraint $J(R_2)K$. Since Θ is path consistent, the constraint $I(R_3)K$ exists, and $\pi(\Theta)$ contains its translation. Again, because (*i* $R_3 \subseteq R_1 \diamond R_2$, (*ii* weak composition computes the *strongest implied relation* given two intervals (or points), and (*iii* $(X \preceq Z)$ would be resolved solely from C and $\mathcal{C}$, it must be the case that $(X \shortparallel Z) \in \pi(\Theta)$, and again, no new clause can be resolved.

- Finally, suppose that C is an instance of some other axiom. C cannot be an instance of the reflexivity axiom because it would only resolve itself. If C is an instance of some antisymmetry axiom, then both C_1 and C_2 must refer to the same two endpoints as C, that is, they must belong to the translation of the same constraint $I(R)J$; but since $\pi(\Theta)$ is explicit, the resulting clause must already be included. If C is the instance of some weakening axiom, or the instance of some incompatibility axiom, then C_1 must refer to the same two endpoints as C, and again, the same argument applies.

Therefore, we can conclude that no deduction can be performed by positive unit resolution on the translation of a path consistent instance; in particular, the empty clause cannot be derived. $\square$

Corollary 4.4.6. *Path Consistency decides* SAT($\mathcal{BA}_{Horn}$).

Thus, $\mathcal{BA}_{Horn}$ is a weak PC-tractable subalgebra of the $\mathcal{BA}$ that extends $\mathcal{BA}_{convex}$. It would be natural, at this point, to discuss the maximality of $\mathcal{BA}_{Horn}$. To this end, recall the $\mathcal{IA}$ fragments $\mathcal{N}_{1,\dots,6}$ of [63] introduced in Section 4.3.1, which imply the hardness for NP of the consistency problem for a weak subalgebra of the $\mathcal{IA}$ that contains any of them. A fragment of the $\mathcal{BA}$ that contains one of the $\mathcal{N}_i$ cannot be immediately shown to be intractable with the same reasoning; however, this becomes possible if such a fragment also includes the relation $\top_{\mathcal{IA}}$. It so happens that every weak extension of $\mathcal{BA}_{Horn}$ is either $\mathcal{BA}$ or contains some of the sets $\mathcal{N}_i$ and the relation $\top_{\mathcal{IA}}$, and it is therefore intractable.

Theorem 4.4.7. $\mathcal{BA}_{Horn}$ *is a maximal tractable weak subalgebra of the* $\mathcal{BA}$.

Proof. By computer-assisted enumeration. $\square$

The question we pose now is whether $\mathcal{BA}_{Horn}$ is unique, or there are more fragments of the $\mathcal{BA}$ with similar properties.

4.4.3 The Disjunctive Pointisable Fragments

We have already seen how disjunctive point relations can be used to represent interval relations in an alternative way. While for the $\mathcal{BA}_{Horn}$ fragment we had to build an ad-hoc Horn ontology to prove its tractability, in this section we will exploit Broxvall's tractable fragments $\mathcal{T}_{A,\dots,E}$ of the $\mathcal{BPA}^*$ (see Section 4.3.2 to obtain new tractable fragments in the $\mathcal{BA}$. Since some fragments $\mathcal{S} \subseteq \mathcal{BPA}^*$ are infinite, to construct the corresponding $\mathcal{BA}$ fragment $\xi^{-1}(\mathcal{S})$, we consider only the set $\mathcal{S}' = \{r \mid r \in \mathcal{S} \wedge |r| \leq 4\}$, because any $\mathcal{BA}$ relation can be represented as a disjunction of at most four $\mathcal{BPA}$ relations. First, notice how the two algebras $\mathcal{T}_C$ and $\mathcal{T}_D$ can be immediately excluded from our analysis since they do not contain the relation $<$, which is needed to express $I^- < I^+$. Also, it is the case that $\mathcal{BA}_{Horn} = C(\xi^{-1}(\mathcal{T}_E))$: this equivalence, which is also an alternative way to show the tractability of $\mathcal{BA}_{Horn}$ (but not to show its PC-tractability), becomes clear when comparing the point relations that build $\mathcal{T}_E$ (see Tab. 4.3) and our definition of allowed TORD clause. By applying the inverse point mapping on $\mathcal{T}_A = \mathcal{BPA}$, one obtains, as expected, the pointisable fragment $\mathcal{BA}_{point}$. Finally, by applying this mapping to $\mathcal{T}_B$, we get another fragment, which we denote as the *linear fragment*

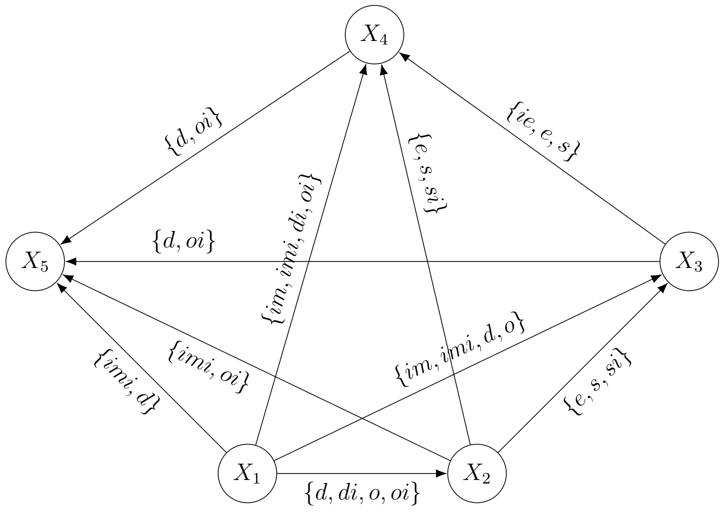


Figure 4.10: A path-consistent, but not consistent, instance of $\mathcal{BA}_{point}$.

of the $\mathcal{BA}$ ($\mathcal{BA}_{lin}$) since all the point relations contained in $\mathcal{T}_B$ are linear, with the exception of $\neq$, notice in particular that $\mathcal{BA}_{lin} \neq \mathcal{IA}$.

Theorem 4.4.8. *The fragments $\mathcal{BA}_{point}$ and $\mathcal{BA}_{lin}$ are tractable weak subalgebras of the $\mathcal{BA}$.*

Proof. As we know, $\mathcal{BA}_{point}$ and $\mathcal{BA}_{lin}$ are the sets of the relations that can be point-mapped to, respectively, $\mathcal{T}_A$ and $\mathcal{T}_B$. Since ξ operates in constant time, we can transform any instance of $\mathcal{BA}_{point}$ and $\mathcal{BA}_{lin}$ in polynomial time to an equivalent instance of, respectively, $\mathcal{T}_A$ and $\mathcal{T}_B$. □

The tractability of $\mathcal{BA}_{point}$ and $\mathcal{BA}_{lin}$ clearly does not imply their PC-tractability. In particular, it is possible to prove the following result.

Theorem 4.4.9. *$\mathcal{BA}_{point}$ is not* PC*-tractable.*

Proof. The fact that *Path Consistency* is incomplete for checking the consistency of instances of $\mathcal{BA}_{point}$ can be shown by proving the existence of at least one inconsistent, but PC-consistent, instance. One such example is given in Figure 4.10. □

Whether $\mathcal{BA}_{lin}$ instances can be checked by *Path Consistency* or not is an open problem; extensive search for counterexamples gave negative results. Just as we have only a partial picture of the PC-tractability of disjunctive fragments, we have only a partial picture of their maximality. By computer enumeration it can be shown that there are precisely five supersets of $\mathcal{BA}_{point}$ and nine supersets of $\mathcal{BA}_{lin}$ whose tractability is unknown.

Every extension of $\mathcal{BA}_{point}$ (resp. $\mathcal{BA}_{lin}$) is either intractable or belongs to the extension graph shown in Figure 4.11 (resp. Figure 4.12).

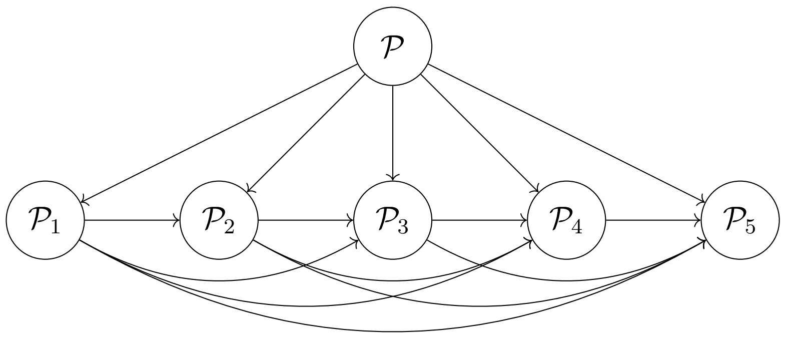


Figure 4.11: Extension graph of $\mathcal{BA}_{point}$ (denoted, here, by $\mathcal{P}$).

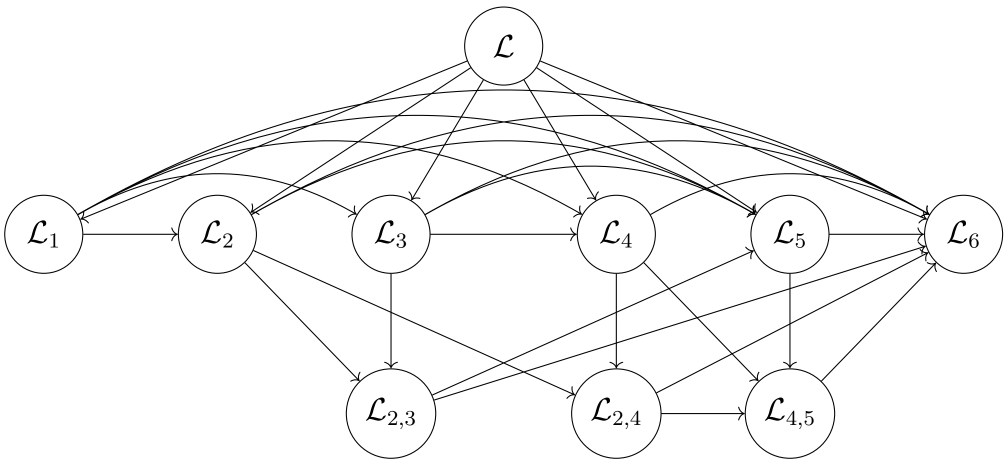


Figure 4.12: Extension graph of $\mathcal{BA}_{lin}$ (denoted, here, by $\mathcal{L}$).

Definition 4.4.3 (Extension graph)**.** Given a relation algebra $\mathcal{A}$, we call *extension graph* of a fragment $\mathcal{S} \subseteq \mathcal{A}$ a directed graph $G = (V, E)$ that represents the set of the least generated subalgebras obtained by extending $\mathcal{S}$ with some relation $R \notin \mathcal{S}$: vertices represent subalgebras, and edges indicate that one set can be extended into another, and can be labelled with the relation R. An extensions graph is called *interesting* if its (in)tractability is not trivial.

It is possible to verify the following. While in the case of $\mathcal{BA}_{point}$ such supersets form a chain w.r.t. set containment, in the case of $\mathcal{BA}_{lin}$ the situation is more complex, with six supersets that form three chains, and three supersets formed by combinations of other supersets. Since tractability is inherited by subsets, and intractability is inherited by supersets, proving the intractability of, say, $\mathcal{P}_1$ would imply that also $\mathcal{P}_{2,\ldots,5}$ are intractable, and therefore $\mathcal{BA}_{point}$ is maximal, while proving the tractability of, say, $\mathcal{L}_6$ would mean that it is also maximal, since all its extension contain at least one of the intractable sets $\mathcal{N}_{1,\ldots,6}$.

4.5 Experiments

We experimentally evaluate the usefulness of $\mathcal{BA}_{convex}$ and $\mathcal{BA}_{Horn}$ fragments as a heuristic for checking the consistency of a $\mathcal{BA}$-network.

4.5.1 Consistency of Temporal Constraint Network

Practical algorithms for approaching NP-COMPLETE problems often use backtracking algorithms based on constraint propagation. We designed a simple algorithm based on encoding the temporal network into a CSP using the classical approach by Condotta et al. [51]. Given a $\mathcal{BA}$-network $N = (V, E)$, its dual CSP is a triple $\mathscr{P} = \langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where:

(i) the set $\mathcal{V}$ contains a variable ν_{XY} for each pair of variables X, Y in V;

(ii) the set $\mathcal{D}$ contains a domain D_{XY} for each variable ν_{XY}, and

(iii) $\mathcal{C}$ contains a binary constraint $inverse(\nu_{XY}, \nu_{YX})$ for each pair of nodes $X, Y \in V$, satisfied by all pairs (r, r^{-1}), where $r \in \mathcal{BA}_{basic}$, and a ternary constraint $composition(\nu_{XY}, \nu_{YZ}, \nu_{XZ})$ for each triple of nodes $X, Y, Z \in V$, which encodes the composition table and is satisfied by all triples $(r_1, r_2, r_3) \in \mathcal{BA}^3_{basic}$ such that $r_3 \in r_2 \circ r_1$.

As noted by Condotta et al. [51], enforcing (generalized) arc consistency on the problem $\mathscr{P}$ is equivalent to enforcing path consistency on the original $\mathcal{BA}$-network. Since, as we know, both path consistency and (generalized) arc consistency are incomplete algorithms for general $\mathcal{BA}$-networks, a backtracking search is applied, and to each node of the search tree, (generalized) arc consistency is enforced.

Algorithm 7 sketches the generic schema of a backtracking algorithm to decide consistency of a network (see [153]). In Algorithm 7, the family of sets *Split* plays a key role. When solving the general CSP, without exploiting any tractable fragment of the $\mathcal{BA}$, *Split* can be thought of as containing all singletons, each one of them corresponding to a single $\mathcal{BA}_{basic}$-relation. Therefore, the technique to solve a problem $\mathscr{P}$ consists of simply choosing a variable, substituting its domain with one of its components creating a new problem $\mathscr{P}'$, and recursively solving $\mathscr{P}'$. This algorithm is correct because the search terminates with a node with basic relations only, for which path consistency is a complete method. When a bigger fragment for which enforcing path consistency is known to be complete for consistency, we can exploit it by setting *Split* to be the family of its relations; in such a case Algorithm 7 stops the search even if the domain of some of the CSP variables is not a singleton, obtaining, *de facto*, a smaller branching factor of the search tree.

In order to minimize the branching factor, among the (possibly many) ways to partition a domain, it makes sense to choose one with minimal cardinality. Unfortunately, establishing if a set can be partitioned into smaller sets taken from

Algorithm 7 Backtracking algorithm for deciding consistency of temporal constraint network

```
function CONSISTENT($\mathscr{P}$, Split)
    enforce generalized arc consistency on $\mathscr{P}$
    if there is a variable $\nu_{XY}$ such that $D_{XY} = \emptyset$ then
        return false
    choose an unprocessed variable $\nu_{XY}$ such that $D_{XY} \notin Split$
    if there is no such variable then
        return true
    $\{D_1, \ldots, D_p\}$=PARTITION($D_{XY}$, Split)
    for all $D_i \in \{D_1, \ldots, D_p\}$ do
        $\mathscr{P}' = \mathscr{P}_{D_{XY}/D_i}$
        if CONSISTENT($\mathscr{P}'$, Split) then
            return true
    return false
```

some family of sets is an instance of the *set partitioning problem*, which is NP-COMPLETE on its own (if we consider the size of the set of base relations as variable); this step is encoded into a function PARTITION(D, *Split*) at line 8. Since this problem should be solved in every node of the search tree, a quick implementation is mandatory to obtain reasonable efficiency, possibly at the expense of optimality.

In our experiments, we store the tractable fragments (sets of relations) into a *trie*; the data structure turned out to be efficient in practice despite the access time depends on the order in which the elements of a set are stored. We also decided to use a greedy method to quickly provide a possibly non-optimal partitioning (note that it is not necessary for PARTITION(D, *Split*) to return the complete partitioning, as each of the sets $D_1, \ldots, D_p$ can be generated on demand).

4.5.2 Experimental setting and results

As witnessed by many previous works [153, 159, 194, 195, 185], it is difficult to find a suitable set of benchmarks, so in most of the works the authors were able to experiment only with random instances. To generate a random set of instances, we used a (modification of) technique suggested by Renz and Nebel [173] that consists of the following steps. Given a number n of nodes, an average density d and a probability p, we generate a random instance as follows:

(i) we generate a graph with n nodes, and select $(d \cdot n \cdot (n-1))/2$ edges at random;

(ii) for each selected edge (s, t), we generate a $\mathcal{BA}$-relation R by selecting, with probability p, each $\mathcal{BA}_{basic}$-relation to be inserted in R, and

(iii) to each non-selected edge (s, t), we assign the universal relation.

Our experiments aim to assess the improvement of the backtracking algorithm when the $\mathcal{BA}_{convex}$ and $\mathcal{BA}_{Horn}$ fragments are used as *Split* heuristics as opposed to using basic relations only. Algorithm 7 was implemented in the constraint logic programming system ECL^{i}PS^e [184], using the libraries CLP(FD) and PROPIA [127], which provides a very general and declarative way to implement new constraints, although we are aware that more efficient implementations could be possible. The objective of the experimentation was discussing the relative improvement given when the $\mathcal{BA}_{convex}$ and $\mathcal{BA}_{Horn}$ fragments are used as *Split* heuristics, rather than evaluating the absolute performances of our implementation.

All experiments were run on an Intel® Xeon® E5-2630 v3 CPU @ 2.4GHz running ECL^{i}PS^e v. 7.0, build #54 on CentOS Linux 7, using only one core and with 1GB of reserved memory; in the results shown in the following, a timeout was counted as the time limit of 600 seconds, thus the timing results can be thought of as a lower bound on the (unknown) real running time.

We generated a set of instances varying the number of nodes n and the density d, while keeping the probability of a constraint between any two variables $p = \frac{1}{2}$. We removed from the dataset the instances for which all the considered algorithms ran out of time; still we ensured that for each pair (n, d) we had at least 100 instances.

Figure 4.13 shows the running time of the backtracking algorithm varying the number of nodes of the network while fixing the density $d = 70\%$. In Figure 4.14 the number of nodes in the network is fixed $n = 16$ while the density is varied. The shape of the curves shows the phase transition: low density networks are easily satisfiable, while in high density networks the unsatisfiability is easily provable. Note that the new fragments improve the performances in particular in the hardest region, at a density between 70% and 80%, in which both satisfiability and unsatisfiability are hard to prove.

The results show that the use of $\mathcal{BA}_{convex}$ fragment manages to reduce the average computation time required to verify the consistency of the $\mathcal{BA}$-network by 16% while the $\mathcal{BA}_{Horn}$ fragment by 27%. Finally, Figure 4.15 shows computation time of 2700 random networks, varying the number of nodes n from 15 to 20 and varying the constraint density d from 55% to 100%. The results show that both the use of $\mathcal{BA}_{convex}$ and $\mathcal{BA}_{Horn}$ fragments leads to reduction in the computation time required to verify the consistency of the $\mathcal{BA}$-network.

It is also worth noting that the use of the $\mathcal{BA}_{convex}$ and $\mathcal{BA}_{Horn}$ fragments results in a significant reduction in the number of timeouts. Among the tested instances, $\mathcal{BA}_{basic}$ incurred in timeout on 160 instances, while $\mathcal{BA}_{convex}$ on 54 and $\mathcal{BA}_{Horn}$ on 20. In particular, the use of the $\mathcal{BA}_{Horn}$ fragment provides the best results with a reduction in computation time not only with respect to not using it but also with respect to using the $\mathcal{BA}_{convex}$ fragment.

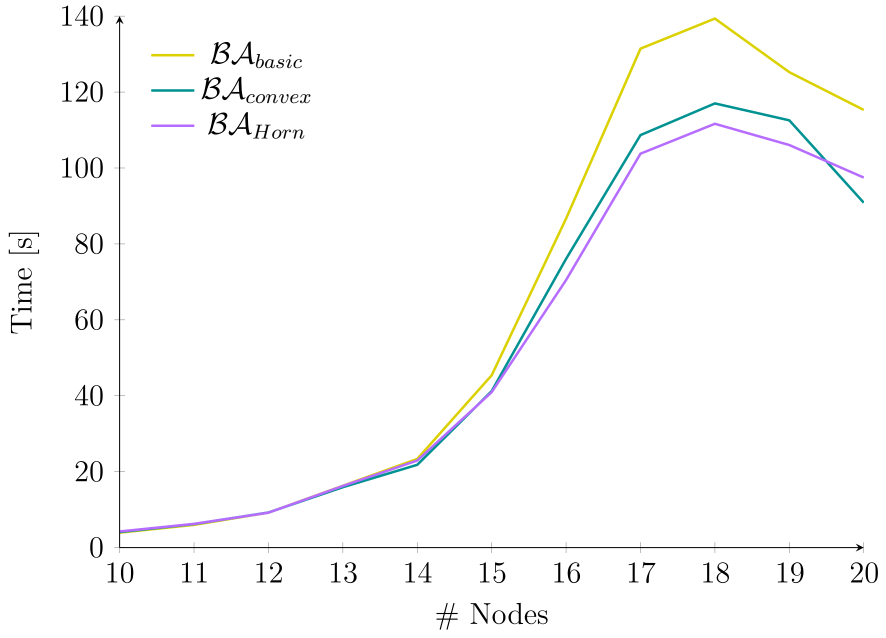


Figure 4.13: Running time of the backtracking algorithm varying the number of nodes n of the network. Each point represents the geometric mean of 100 instances, with density $d = 70\%$. Different lines represent different fragments as *Split* set.

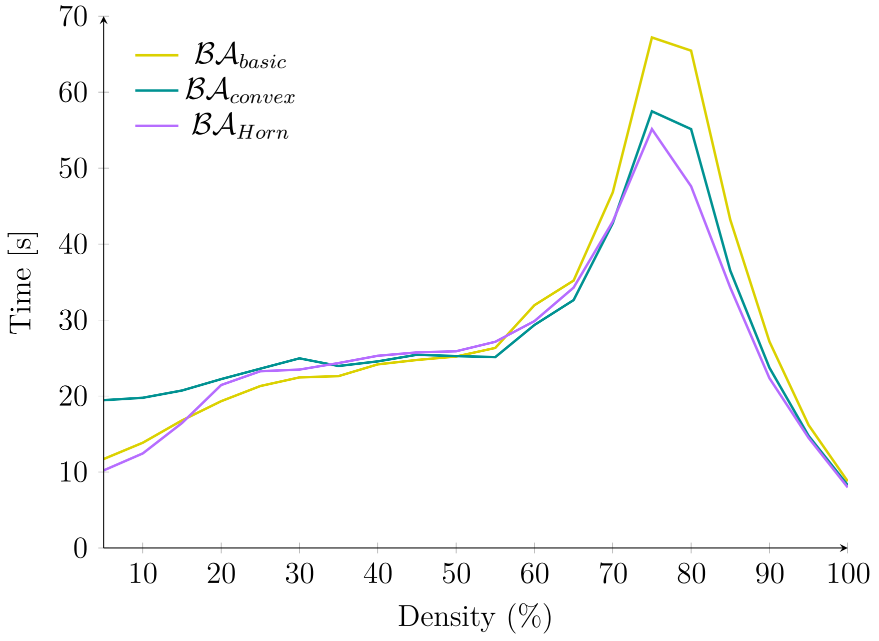


Figure 4.14: Running time of the backtracking algorithm varying the density d of the network. Each point represents the geometric mean of 50 instances, with number of nodes $n = 16$. Different lines represent different fragments as *Split* set.

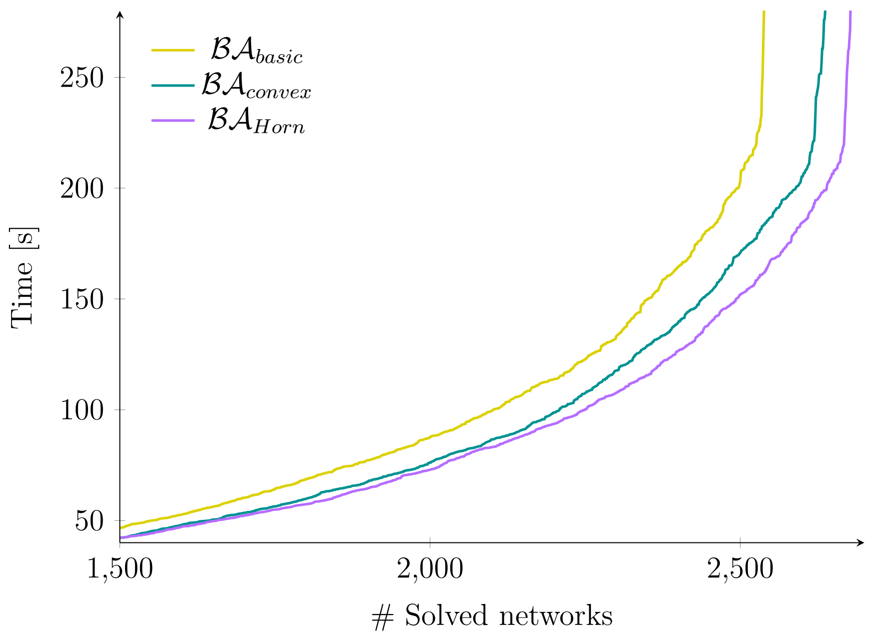


Figure 4.15: Cactus plot showing the number of solved instances varying the solving time. Instances have been generated with a number n of nodes varying from 15 to 20 and a constraint density d varying from 55% to 100%. Different lines represent different fragments as *Split* set in backtracking algorithm.

5 Conclusion and Future Works

In this thesis, we proposed new efficient reasoning algorithms for solving route planning and qualitative temporal problems in Constraint Programming. The implementation of all the proposed algorithms is based on a declarative approach using the Constraint Logic Programming system ECL^{i}PS^e.

In Chapter 3 we presented the Euclidean Traveling Salesperson Problem (ETSP). The ETSP is a special case, of the most famous Traveling Salesperson Problem (TSP), in which each node is identified by its coordinates on the plane and the Euclidean distance is used as cost function. No specialized CP pruning algorithms had been proposed before for Euclidean TSPs, and the usual way to approach Euclidean TSPs is to compute the distance matrix and approach the problem as a general TSP.

We proposed to use the geometric information present in ETSP instances to provide additional pruning with respect to the techniques already available in CP. We presented the implementation of two new redundant constraints called: `nocrossing` and `clockwise`. This, to the best of our knowledge, is the first attempt to exploit such additional information to prune the search space in CP.

We showed that the pruning we perform is orthogonal with respect to that obtained by Benchimol et al. [27] in their seminal work, and that adding reasoning on geometrical properties can further reduce the running time. Despite the results that we have achieved and presented in this thesis, our approaches are still not competitive with Concorde, however, they are useful to improve constraint propagation in CP, an important solving technique. Moreover, Concorde can also be applied to the TSP and not to its variants, while CP formulations can easily handle situations where additional constraints must be considered.

As future work we plan to apply extensions of the proposed techniques in the Euclidean VRP, Euclidean Generalized TSP, and other similar problems. For example, in the optimal solution of a Euclidean Vehicle Routing Problem, the tour of each vehicle is not self-intersecting (although it can cross the path of other vehicles). The proposed techniques could also be generalised for application in metric TSPs as long as the geometric coordinate information is available.

One limitation of the current work is that it is not universally applicable to all TSP variants, such as the TSP with Time Windows, since its optimal solutions may contain crossings; but crossings avoidance could also be interesting *per se* in some applications. It should also be noted that if we consider the *human factor*, routes without crossings are easier to memorise and therefore more easily accepted by both workers and management staff.

Another future work concerns the extension of what is presented here in CLP to other declarative approaches such as MiniZinc[1] and Answer Set Programming (ASP).

We also studied the propagation efficiency of newly introduced `nocrossing` constraints by collecting data while solving ETSPs instances.

Then we proposed to use supervised machine learning techniques to predict and select only the set of `nocrossing` constraints that are useful for the considered instance of the problem in order to increase the overall solving performance.

Some improvement could be achieved by expanding the experimental dataset, i.e., by running experiments in more instances to widen the available data about number of activations and pruning of the `nocrossing` constraints. When expanding the dataset of experiments, the use of structured instances and their generalisations could be particularly interesting. Note also that only one search heuristic was employed in the current dataset, namely *max regret* [46]. Since max regret is a dynamic search heuristic, it might be the case that changing the set of `nocrossing` constraints, the search strategy radically changes the exploration of the search tree, possibly shuffling the order in which `nocrossing` constraints are activated, and making effective some constraints that were not and vice-versa. So, a more precise classifier could be obtained by generating datasets with different search strategies.

Other possible new directions might be to consider predicting the actual ratio of pruning power versus activations, rather than two classes, run experiments imposing only those `nocrossing` constraints whose ratio is predicted to be higher than a predetermined threshold, and experimentally find the best possible value of the threshold. This shifts the machine learning step from classification to regression, which, on the other hand, may be more challenging. One could also move from supervised learning to reinforcement learning techniques and thus avoid collecting a training set; but this needs a complete change in the definition of the machine learning problem both in terms of inputs, outputs, metrics to consider, and number of calls to the TSP solver. Furthermore, the use of reinforcement learning has the additional advantage of being able to consider the interaction of

[1]https://www.minizinc.org/

several constraints applied at the same time and thus also solve issues related to the use of different search strategies.

Finally, instead of learning a classifier that selects the set of `nocrossing` constraints to be imposed a priori, more dynamic strategies could be used, such as removing during search the `nocrossing` constraints that result less effective because they have not obtained significant pruning in recent activations. One source of inspiration could be the strategies used in SAT solvers to forget some of the nogoods [87].

In Chapter 4 we addressed Branching Interval Algebra ($\mathcal{BA}$) which is the natural branching-time generalization of Allen's Interval Algebra, and it has many potential applications in different areas of Artificial Intelligence (AI). As in the linear case, the consistency problem of Branching Algebra is NP-HARD, and studying its tractable fragments is an interesting problem. In the linear case, every tractable fragment of the full algebra is known, while in the branching case the entire landscape of tractable fragments is still unknown.

Branching Algebra has been introduced in [168], where it has been proven that the consistency problem for the subset that includes only basic relations is tractable.

We systematically explored the (PC-)tractability of fragments of the Branching Algebra. In particular, we identify four interesting fragments named: $\mathcal{BA}_{convex}$, $\mathcal{BA}_{point}$, $\mathcal{BA}_{lin}$, $\mathcal{BA}_{Horn}$. The known tractable fragments of $\mathcal{BA}$ are reported in Fig. 5.1, and can be summarized as follows:

- consistency and minimality for $\mathcal{BA}_{convex}$ are both (PC-)tractable, and $\mathcal{BA}_{convex}$ is maximal in terms of PC-tractability of minimality;
- consistency and minimality for $\mathcal{BA}_{Horn}$ are both tractable, but only consistency is PC-tractable in this case, and $\mathcal{BA}_{Horn}$ is maximal w.r.t. tractability;
- consistency and minimality for $\mathcal{BA}_{point}$ are both tractable, but not PC-tractable, and maximality w.r.t. tractability is unknown;
- consistency and minimality for $\mathcal{BA}_{lin}$ are also both tractable but both their maximality and their PC-tractability is unknown.

The maximality of $\mathcal{BA}_{point}$ and $\mathcal{BA}_{lin}$ is still an open problem, although we proved some possibly useful partial results in this sense. Also, $\mathcal{BA}_{point}$ is not included in $\mathcal{BA}_{Horn}$, unlike its linear counterpart, and $\mathcal{BA}_{lin}$ does not even have a linear counterpart, although it is a proper superset of $\mathcal{IA}_{Horn}$, so we suspect that an analysis like the one we did for $\mathcal{BA}_{Horn}$ might bring some new results. Furthermore, $\mathcal{BA}_{lin}$ loses its tractability when extended with any $\mathcal{BA}_{basic}$ relation.

Finally, we design an enhanced version of the classic backtracking consistency algorithm for the full $\mathcal{BA}$ that takes advantage from tractable fragments. We carried out a series of experiments which showed that using $\mathcal{BA}_{convex}$ and $\mathcal{BA}_{Horn}$

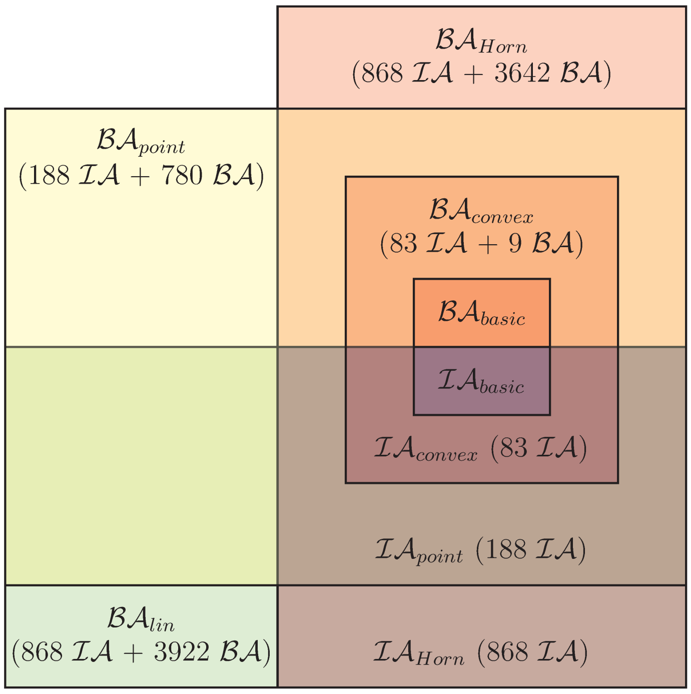

$\mathcal{BA}_{Horn}$
(868 $\mathcal{IA}$ + 3642 $\mathcal{BA}$)
$\mathcal{BA}_{point}$
(188 $\mathcal{IA}$ + 780 $\mathcal{BA}$)
$\mathcal{BA}_{convex}$
(83 $\mathcal{IA}$ + 9 $\mathcal{BA}$)
$\mathcal{BA}_{basic}$
$\mathcal{IA}_{basic}$
$\mathcal{IA}_{convex}$ (83 $\mathcal{IA}$)
$\mathcal{IA}_{point}$ (188 $\mathcal{IA}$)
$\mathcal{BA}_{lin}$
(868 $\mathcal{IA}$ + 3922 $\mathcal{BA}$)
$\mathcal{IA}_{Horn}$ (868 $\mathcal{IA}$)

Figure 5.1: An Euler-Venn diagram representation of the known tractable fragments in the $\mathcal{BA}$, and their counterparts in the $\mathcal{IA}$. For each set, we also show the number of relations it contains (a relation is $\mathcal{IA}$ if it only contains linear basic relations, otherwise it is $\mathcal{BA}$).

fragments as heuristics in the backtraking algorithm manages to reduce the average computation time required to verify the consistency of the $\mathcal{BA}$-network.

What we presented constitutes yet another step towards the complete classification between tractable/intractable fragments of Branching Algebra. At the moment, the Horn fragment is the biggest tractable known fragment. Yet, other, incomparable fragments may exist.

The techniques, and the algorithms, needed to perform this classification for an algebra with 19 relations (much bigger than the $\mathcal{IA}$, with 13 relations only) can be certainly re-used for similar studies in other more complicated algebras, such as the Rectangle Algebra or the Block Algebra, and similar formalism for spatial-temporal reasoning.

References

[1] Martín Abadi et al. “TensorFlow: A System for Large-Scale Machine Learning”. In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation.* OSDI’16. Savannah, GA, USA: USENIX Association, 2016, 265–283. ISBN: 9781931971331.

[2] Richa Agarwala, David L Applegate, Donna Maglott, Gregory D Schuler, and Alejandro A Schäffer. “A fast and scalable radiation hybrid map construction and integration strategy”. In: *Genome Research* 10.3 (2000), pp. 350–364.

[3] Charu Aggarwal. *Neural Networks and Deep Learning - A Textbook.* Springer, 2018. ISBN: 978-3-319-94462-3.

[4] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows - theory, algorithms and applications.* Prentice Hall, 1993. ISBN: 978-0-13-617549-0.

[5] James F. Allen. “Planning as Temporal Reasoning”. In: *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR’91). Cambridge, MA, USA, April 22-25, 1991.* Ed. by James F. Allen, Richard Fikes, and Erik Sandewall. Morgan Kaufmann, 1991, pp. 3–14.

[6] J.F. Allen. “Maintaining Knowledge about Temporal Intervals”. In: *Communications of the ACM* 26.11 (1983), pp. 832–843.

[7] N. Amaneddine and J-F. Condotta. “From Path-Consistency to Global Consistency in Temporal Qualitative Constraint Networks”. In: *15th International Conference on Artificial Intelligence: Methodology, Systems, and Applications.* Vol. 7557. LNCS. 2012, pp. 152–161.

[8] Nouhad Amaneddine and Jean-François Condotta. “From Path-Consistency to Global Consistency in Temporal Qualitative Constraint Networks”. In: Sept. 2012, pp. 152–161. ISBN: 978-3-642-33184-8.

[9] Mario Amrehn, Firas Mualla, Elli Angelopoulou, Stefan Steidl, and Andreas Maier. *The Random Forest Classifier in WEKA: Discussion and New Developments for Imbalanced Data.* 2019. arXiv: `1812.08102 [cs.CV]`.

[10] A.M. Andrew. “Another efficient algorithm for convex hulls in two dimensions”. In: *Information Processing Letters* 9.5 (Dec. 1979), pp. 216–219.

[11] Shoshana Anily, Julien Bramel, and Alain Hertz. “A 5/3-approximation algorithm for the clustered traveling salesman tour and path problems”. In: *Oper. Res. Lett.* 24 (1999), pp. 29–35.

[12] Franklin Antonio. “IV.6 - FASTER LINE SEGMENT INTERSECTION”. In: *Graphics Gems III (IBM Version).* Ed. by DAVID KIRK. San Francisco: Morgan Kaufmann, 1992, pp. 199–202. ISBN: 978-0-12-409673-8.

[13] David Applegate, Robert E. Bixby, Vasek Chvátal, and William J. Cook. "TSP Cuts Which Do Not Conform to the Template Paradigm". In: *Computational Combinatorial Optimization, Optimal or Provably Near-Optimal Solutions.* Ed. by Michael Jünger and Denis Naddef. Vol. 2241. Lecture Notes in Computer Science. Springer, 2001, pp. 261–304. ISBN: 3-540-42877-1.

[14] David L Applegate, Robert E Bixby, Vasek Chvatal, and William J Cook. *The traveling salesman problem: a computational study.* Princeton university press, 2006.

[15] Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003. ISBN: 978-0-521-82583-2.

[16] S. Arora. "Polynomial time approximation schemes for Euclidean TSP and other geometric problems". In: *Proceedings of 37th Conference on Foundations of Computer Science.* 1996, pp. 2–11.

[17] Norbert Ascheuer and Matteo Fischetti. "A polyhedral study of the asymmetric traveling salesman problem with time windows". In: *Networks* 36 (Sept. 2000).

[18] Norbert Ascheuer, Matteo Fischetti, and Martin Grötschel. "Solving the Asymmetric Travelling Salesman Problem with Time Windows by branch-and-cut". In: *Mathematical Programming* 90 (Jan. 2001), pp. 475–506.

[19] G B. Dantzig, D R. Fulkerson, and S M. Johnson. "Solution of a Large-Scale Traveling Salesman Problem". In: *Operations Research* 2 (Jan. 1954), pp. 393–410.

[20] Fahiem Bacchus and Paul van Run. "Dynamic Variable Ordering in CSPs". In: *Principles and Practice of Constraint Programming - CP'95, First International Conference, CP'95, Cassis, France, September 19-22, 1995, Proceedings.* Ed. by Ugo Montanari and Francesca Rossi. Vol. 976. Lecture Notes in Computer Science. Springer, 1995, pp. 258–275.

[21] Egon Balas, Matteo Fischetti, and William Pulleyblank. "The precedence-constraint asymmetric traveling salesman polytope". In: *Math. Program.* 68 (Mar. 1995), pp. 241–265.

[22] N Beldiceanu and E Contejean. "Introducing Global Constraints in CHIP". In: *Math. Comput. Model.* 20.12 (Dec. 1994), pp. 97–123. ISSN: 0895-7177.

[23] Nicolas Beldiceanu and Helmut Simonis. "ModelSeeker: Extracting Global Constraint Models from Positive Examples". In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach.* Ed. by Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi. Vol. 10101. LNCS. Springer, 2016, pp. 77–95. ISBN: 978-3-319-50136-9.

[24] Elena Bellodi, Alessandro Bertagnon, Marco Gavanelli, and Riccardo Zese. "Improving the Efficiency of Euclidean TSP Solving in Constraint Programming by Predicting Effective Nocrossing Constraints". In: *AIxIA 2020 - Advances in Artificial Intelligence - XIXth International Conference of the Italian Association for Artificial Intelligence, Virtual Event, November 25-27, 2020, Revised Selected Papers.* Ed. by Matteo Baldoni and Stefania Bandini. Vol. 12414. Lecture Notes in Computer Science. Springer, 2020, pp. 318–334.

[25] Elena Bellodi, Alessandro Bertagnon, Marco Gavanelli, and Riccardo Zese. "Improving the Efficiency of Euclidean TSP Solving in Constraint Programming by Predicting Effective Nocrossing Constraints". In: *Joint Proceedings of the 8th Italian Workshop on Planning and Scheduling and the 27th International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion co-located with AIxIA 2020, Online Event, November 25-27, 2020.* Ed. by Riccardo De Benedictis et al. Vol. 2745. CEUR Workshop Proceedings. CEUR-WS.org, 2020.

[26] Amir Ben-Dor, Benny Chor, and Dan Pelleg. "RHO—radiation hybrid ordering". In: *Genome Research* 10.3 (2000), pp. 365–378.

[27] Pascal Benchimol, Willem Jan van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. "Improved filtering for weighted circuit constraints". In: *Constraints* 17.3 (2012), pp. 205–233.

[28] Seymour Benzer. "On the topology of the genetic fine structure". In: *Proceedings of the National Academy of Sciences of the United States of America* 45.11 (1959), p. 1607.

[29] Alessandro Bertagnon. "Constraint Programming Algorithms for Route Planning Exploiting Geometrical Information". In: *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020.* Ed. by Francesco Ricca et al. Vol. 325. EPTCS. 2020, pp. 286–295.

[30] Alessandro Bertagnon and Marco Gavanelli. "Improved Filtering for the Euclidean Traveling Salesperson Problem in CLP(FD)". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020.* AAAI Press, 2020, pp. 1412–1419.

[31] Alessandro Bertagnon, Marco Gavanelli, Alessandro Passantino, Guido Sciavicco, and Stefano Trevisani. "The Horn Fragment of Branching Algebra". In: *Proc. of the 27th International Symposium on Temporal Representation and Reasoning.* Vol. 178. LIPIcs. 2020, 5:1–5:16.

[32] Alessandro Bertagnon, Marco Gavanelli, Alessandro Passantino, Guido Sciavicco, and Stefano Trevisani. "Branching interval algebra: An almost complete picture". In: *Information and Computation* 281 (2021), p. 104809. ISSN: 0890-5401.

[33] Alessandro Bertagnon, Marco Gavanelli, Guido Sciavicco, and Stefano Trevisani. "On (Maximal, Tractable) Fragments of the Branching Algebra". In: *Proceedings of the 35th Italian Conference on Computational Logic - CILC 2020, Rende, Italy, October 13-15, 2020.* Ed. by Francesco Calimeri, Simona Perri, and Ester Zumpano. Vol. 2710. CEUR Workshop Proceedings. CEUR-WS.org, 2020, pp. 113–126.

[34] Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi, eds. *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach.* Vol. 10101. LNCS. Springer, 2016. ISBN: 978-3-319-50136-9.

[35] Christian Bessière, Amar Isli, and Gerard Ligozat. "Global Consistency in Interval Algebra Networks: Tractable Subclasses". In: *Proc. of the 12th European Conference on Artificial Intelligence.* Ed. by Wolfgang Wahlster. 1996, pp. 3–7.

[36] Christian Bessière and Jean-Charles Régin. "Arc Consistency for General Constraint Networks: Preliminary Results". In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes.* Morgan Kaufmann, 1997, pp. 398–404.

[37] Christian Bessiere et al. "New Approaches to Constraint Acquisition". In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach.* Ed. by Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O'Sullivan, and Dino Pedreschi. Vol. 10101. LNCS. Springer, 2016, pp. 51–76. ISBN: 978-3-319-50136-9.

[38] Robert G Bland and David F Shallcross. "Large travelling salesman problems arising from experiments in X-ray crystallography: a preliminary report on computation". In: *Operations Research Letters* 8.3 (1989), pp. 125–128.

[39] Daniel G. Bobrow and Bertram Raphael. "New Programming Languages for Artificial Intelligence Research". In: *ACM Comput. Surv.* 6.3 (1974), 153–174. ISSN: 0360-0300.

[40] Ernesto Bonomi and Jean-Luc Lutton. "The N-City Travelling Salesman Problem: Statistical Mechanics and the Metropolis Algorithm". In: *SIAM Review* 26 (Oct. 1984), p. 551.

[41] Jakob Bossek. "netgen: Network generator for combinatorial graph problems". In: *R package version* 1 (2015).

[42] Leo Breiman. "Random Forests". In: *Mach. Learn.* 45.1 (Oct. 2001), 5–32. ISSN: 0885-6125.

[43] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, 1984. ISBN: 0-534-98053-8.

[44] M. Broxvall. "The Point Algebra for Branching Time Revisited". In: *Proc. of Advances in Artificial Intelligence.* Vol. 2174. LNCS. Springer, 2001, pp. 106–121.

[45] Maurice Bruynooghe. "Solving Combinatorial Search Problems by Intelligent Backtracking". In: *Inf. Process. Lett.* 12.1 (1981), pp. 36–39.

[46] Yves Caseau and François Laburthe. "Solving Small TSPs with Constraints". In: *Logic Programming, Proc. of the Fourteenth International Conference on Logic Programming, Leuven, Belgium, July 8-11, 1997.* Ed. by Lee Naish. MIT Press, 1997, pp. 316–330. ISBN: 0-262-64035-X.

[47] Augustin Cauchy et al. "Méthode générale pour la résolution des systemes d'équations simultanées". In: *Compte rendu des séances de l'académie des sciences* 25.1847 (1847), pp. 536–538.

[48] Nicos Christofides. *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem.* Management sciences research report. Defense Technical Information Center, 1976.

[49] Alain Colmerauer. "Prolog-II Reference Manual". In: *Technical Report.* Université Aix - Marseille II, 1982.

[50] Alain Colmerauer. “An Introduction to Prolog III”. In: *Commun. ACM* 33.7 (1990), pp. 69–90.

[51] J.F. Condotta, D. D’Almeida, C. Lecoutre, and L. Saïs. “From qualitative to discrete constraint networks”. In: *Proc. of the Workshop on Qualitative Constraint Calculi.* 2006, pp. 54–64.

[52] David R Cox, Margit Burmeister, E Roydon Price, Suwon Kim, and Richard M Myers. “Radiation hybrid mapping: a somatic cell genetic method for constructing high-resolution maps of mammalian chromosomes”. In: *Science* 250.4978 (1990), pp. 245–250.

[53] G. A. Croes. “A Method for Solving Traveling-Salesman Problems”. In: *Operations Research* 6.6 (1958), pp. 791–812. ISSN: 0030364X, 15265463.

[54] S. Darabi, S.C.C. Blom, and M. Huisman. “A Verification Technique for Deterministic Parallel Programs”. In: *Proc. of the 9th International Symposium on NASA Formal Methods.* Vol. 10227. LNCS. 2017, pp. 247–264.

[55] J. Davis and M. Goadrich. “The relationship between Precision-Recall and ROC curves”. In: *European Conference on Machine Learning (ECML 2006).* ACM, 2006.

[56] Rina Dechter. “Learning While Searching in Constraint-Satisfaction-Problems”. In: *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, USA, August 11-15, 1986. Volume 1: Science.* Ed. by Tom Kehler. Morgan Kaufmann, 1986, pp. 178–185.

[57] Rina Dechter. “Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition”. In: *Artif. Intell.* 41.3 (1990), pp. 273–312.

[58] Rina Dechter. *Constraint processing.* Elsevier Morgan Kaufmann, 2003. ISBN: 978-1-55860-890-0.

[59] Vladimir G. Deineko, René van Dal, and Günter Rote. “The Convex-Hull-and-Line Traveling Salesman Problem: A Solvable Case”. In: *Inf. Process. Lett.* 51.3 (1994), pp. 141–148.

[60] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, and Françoise Berthier. “The Constraint Logic Programming Language CHIP”. In: *Proceedings of the International Conference on Fifth Generation Computer Systems, FGCS 1988, Tokyo, Japan, November 28-December 2, 1988.* OHMSHA Ltd. Tokyo and Springer-Verlag, 1988, pp. 693–702.

[61] Grégoire Dooms, Yves Deville, and Pierre Dupont. “CP(Graph): Introducing a Graph Computation Domain in Constraint Programming”. In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings.* Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, 2005, pp. 211–225. ISBN: 3-540-29238-1.

[62] Thomas Drakengren and Peter Jonsson. “Twenty-one large tractable subclasses of Allen’s algebra”. In: *Artificial Intelligence* 93.1 (1997), pp. 297 –319.

[63] Thomas Drakengren and Peter Jonsson. “A Complete Classification of Tractability in Allen’s Algebra Relative to Subsets of Basic Relations”. In: *Artificial Intelligence* 106.2 (1998).

[64] Herbert Edelsbrunner, Günter Rote, and Emo Welzl. “Testing the Necklace Condition for Shortest Tours and Optimal Factors in the Plane”. In: *Theor. Comput. Sci.* 66.2 (1989), pp. 157–180.

[65] E.A. Emerson. “Temporal and Modal Logic”. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B).* MIT Press, 1990, pp. 995–1072.

[66] E.A. Emerson and J.Y. Halpern. “Decision Procedures and Expressiveness in the Temporal Logic of Branching Time”. In: *Journal of Computer Systems Science* 30.1 (1985), pp. 1–24.

[67] Susan L. Epstein, Eugene C. Freuder, Richard Wallace, Anton Morozov, and Bruce Samuels. “The Adaptive Constraint Engine”. In: *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming.* CP ’02. Berlin, Heidelberg: Springer-Verlag, 2002, 525–542. ISBN: 3540441204.

[68] Jean-Guillaume Fages and Xavier Lorca. “Improving the Asymmetric TSP by Considering Graph Structure”. In: *CoRR* abs/1206.3437 (2012). arXiv: `1206.3437`.

[69] Jean-Guillaume Fages, Xavier Lorca, and Louis-Martin Rousseau. “The salesman and the tree: the importance of search in CP”. In: *Constraints* 21.2 (2016), pp. 145–162.

[70] Matteo Fischetti, Juan José Salazar González, and Paolo Toth. “A Branch-And-Cut Algorithm for the Symmetric Generalized Traveling Salesman Problem”. In: *Operations Research* 45.3 (1997), pp. 378–394. ISSN: 0030364X, 15265463.

[71] Matteo Fischetti and Paolo Toth. “An additive bounding procedure for the asymmetric travelling salesman problem”. In: *Math. Program.* 53 (1992), pp. 173–197.

[72] M. M. Flood. “The traveling-salesman problem”. In: *Operations Research* 4 (1956).

[73] Filippo Focacci, Andrea Lodi, and Michela Milano. “A Hybrid Exact Algorithm for the TSPTW”. In: *INFORMS Journal on Computing* 14.4 (2002), pp. 403–417.

[74] Filippo Focacci, Andrea Lodi, and Michela Milano. “Embedding Relaxations in Global Constraints for Solving TSP and TSPTW”. In: *Ann. Math. Artif. Intell.* 34.4 (2002), pp. 291–311.

[75] Kathryn Glenn Francis and Peter J. Stuckey. “Explaining circuit propagation”. In: *Constraints* 19.1 (2014), pp. 1–29.

[76] Eugene C. Freuder. “Synthesizing Constraint Expressions”. In: *Commun. ACM* 21.11 (1978), 958–966. ISSN: 0001-0782.

[77] Alan M. Frisch, Christopher Jefferson, and Ian Miguel. “Symmetry Breaking as a Prelude to Implied Constraints: A Constraint Modelling Pattern”. In: ECAI’04. Valencia, Spain: IOS Press, 2004, 171–175. ISBN: 9781586034528.

[78] Daniel Frost and Rina Dechter. “Dead-End Driven Learning”. In: *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.* Ed. by Barbara Hayes-Roth and Richard E. Korf. AAAI Press / The MIT Press, 1994, pp. 294–300.

[79] Daniel Frost and Rina Dechter. “Look-Ahead Value Ordering for Constraint Satisfaction Problems”. In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1.* IJCAI’95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, 572–578. ISBN: 1558603638.

[80] Thom W. Frühwirth and Slim Abdennadher. *Essentials of constraint programming.* Cognitive Technologies. Springer, 2003. ISBN: 978-3-540-67623-2.

[81] M. R. Garey, R. L. Graham, and D. S. Johnson. “Some NP-complete Geometric Problems”. In: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing.* STOC ’76. Hershey, Pennsylvania, USA: ACM, 1976, pp. 10–22.

[82] John Gaschnig. “Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems”. In: *Proceedings of the Second Canadian Conference on Artificial Intelligence.* 1978, pp. 268–277.

[83] Pieter Andreas Geelen. “Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems”. In: *Proceedings of the 10th European Conference on Artificial Intelligence.* ECAI ’92. Vienna, Austria: John Wiley & Sons, Inc., 1992, 31–35. ISBN: 0471936081.

[84] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. “Data Structures for Generalised Arc Consistency for Extensional Constraints”. In: *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada.* AAAI Press, 2007, pp. 191–197.

[85] Ian P. Gent, Lars Kotthoff, Ian Miguel, and Peter Nightingale. “Machine learning for constraint solver design – A case study for the alldifferent constraint”. In: *CoRR* abs/1008.4326 (2010). arXiv: `1008.4326`.

[86] Ian P. Gent, Ewan MacIntyre, Patrick Presser, Barbara M. Smith, and Toby Walsh. “An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem”. In: *Principles and Practice of Constraint Programming.* Ed. by Eugene C. Freuder. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 179–193. ISBN: 978-3-540-70620-5.

[87] Ian P. Gent, Ian Miguel, and Neil C. A. Moore. “An empirical study of learning and forgetting constraints”. In: *AI Commun.* 25.2 (2012).

[88] Carmen Gervet. “New structures of symbolic constraint objects: sets and graphs”. In: *Third Workshop on Constraint Logic Programming (WCLP’2003).* 1993.

[89] Matthew L. Ginsberg. “Dynamic Backtracking”. In: *J. Artif. Intell. Res.* 1 (1993), pp. 25–46.

[90] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. "Search Lessons Learned from Crossword Puzzles". In: *Proceedings of the 8th National Conference on Artificial Intelligence. Boston, Massachusetts, USA, July 29 - August 3, 1990, 2 Volumes.* Ed. by Howard E. Shrobe, Thomas G. Dietterich, and William R. Swartout. AAAI Press / The MIT Press, 1990, pp. 210–215.

[91] Matthew L. Ginsberg, Michael Frank, Michael P. Halpin, and Mark C. Torrance. "Search Lessons Learned from Crossword Puzzles". In: *Proceedings of the Eighth National Conference on Artificial Intelligence - Volume 1.* AAAI'90. Boston, Massachusetts: AAAI Press, 1990, 210–215. ISBN: 026251057X.

[92] David E. Goldberg. "Genetic Algorithms in Search Optimization and Machine Learning". In: 1988.

[93] Solomon W. Golomb and Leonard D. Baumert. "Backtrack Programming". In: *J. ACM* 12.4 (1965), 516–524. ISSN: 0004-5411.

[94] Martin Golumbic and Ron Shamir. "Complexity and Algorithms for Reasoning about Time: A Graph-Theoretic Approach". In: *Journal of the ACM* 40 (Apr. 1996), pp. 1108–1133.

[95] Martin Charles Golumbic and Ron Shamir. "Complexity and Algorithms for Reasoning about Time: A Graph-Theoretic Approach". In: *J. ACM* 40.5 (1993), pp. 1108–1133.

[96] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016.

[97] Stephen J Goss and Henry Harris. "New method for mapping genes in human chromosomes". In: *Nature* 255.5511 (1975), pp. 680–684.

[98] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. "Optimal control of plotting and drilling machines: a case study". In: *Zeitschrift für Operations Research* 35.1 (1991), pp. 61–84.

[99] Nili Guttmann-Beck, Refael Hassin, Samir Khuller, and Balaji Raghavachari. "Approximation Algorithms with Bounded Performance Guarantees for the Clustered Traveling Salesman Problem". In: *Foundations of Software Technology and Theoretical Computer Science, 18th Conference, Chennai, India, December 17-19, 1998, Proceedings.* Ed. by Vikraman Arvind and Ramaswamy Ramanujam. Vol. 1530. Lecture Notes in Computer Science. Springer, 1998, pp. 6–17.

[100] Robert M. Haralick and Gordon L. Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial Intelligence* 14.3 (1980), pp. 263–313. ISSN: 0004-3702.

[101] M. Held and R. M. Karp. "The traveling-salesman problem and minimum spanning trees". In: *Operations Research* 18 (1970), pp. 1138–1162.

[102] Keld Helsgaun. "An effective implementation of the Lin-Kernighan traveling salesman heuristic". In: *European Journal of Operational Research* 126.1 (2000), pp. 106–130.

[103] Lawrence Henschen and Larry Wos. "Unit Refutations and Horn Sets". In: *Journal of the ACM* 21 (Oct. 1974), pp. 590–605.

[104] Pascal Van Hentenryck. *Constraint satisfaction in logic programming.* Logic programming. MIT Press, 1989. ISBN: 978-0-262-08181-8.

[105] Robin Hirsch. “Expressive Power and Complexity in Algebraic Logic”. In: *Journal of Logic and Computation* 7 (Apr. 1997).

[106] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. “Algorithm runtime prediction: Methods & evaluation”. In: *Artif. Intell.* 206 (2014), pp. 79–111.

[107] Nicolas Isoart and Jean-Charles Régin. “Integration of Structural Constraints into TSP Models”. In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2019, pp. 284–299.

[108] J. Jaffar and J.-L. Lassez. “Constraint Logic Programming”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* POPL ’87. Munich, West Germany: Association for Computing Machinery, 1987, 111–119. ISBN: 0897912152.

[109] Joxan Jaffar and Michael J. Maher. “Constraint Logic Programming: A Survey”. In: *J. Log. Program.* 19/20 (1994), pp. 503–581.

[110] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. “The CLP(R) Language and System”. In: *ACM Trans. Program. Lang. Syst.* 14.3 (1992), 339–395. ISSN: 0164-0925.

[111] David S Johnson and Lyle A McGeoch. “Experimental analysis of heuristics for the STSP”. In: *The traveling salesman problem and its variations.* Springer, 2007, pp. 369–443.

[112] Kees Jongens and Ton Volgenant. “The symmetric clustered traveling salesman problem”. In: *European Journal of Operational Research* 19.1 (1985), pp. 68–75. ISSN: 0377-2217.

[113] Antoine Jouglet and Jacques Carlier. “Dominance rules in combinatorial optimization problems”. In: *European Journal of Operational Research* 212.3 (2011), pp. 433–444. ISSN: 0377-2217.

[114] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *Complexity of Computer Computations.* Ed. by R.E. Miller, J.W. Thatcher, and J.D. Bohlinger. The IBM Research Symposia Series. Boston, MA: Springer, 1972, pp. 85–103.

[115] George Katsirelos and Fahiem Bacchus. “Generalized NoGoods in CSPs”. In: *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA.* Ed. by Manuela M. Veloso and Subbarao Kambhampati. AAAI Press / The MIT Press, 2005, pp. 390–396.

[116] George Katsirelos and Toby Walsh. “A Compression Algorithm for Large Arity Extensional Constraints”. In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings.* Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 379–393.

[117] Latife Genç Kaya and John N. Hooker. “A Filter for the Circuit Constraint”. In: *Principles and Practice of Constraint Programming - CP 2006.* Ed. by Frédéric Benhamou. Vol. 4204. Lecture Notes in Computer Science. Springer, 2006, pp. 706–710. ISBN: 3-540-46267-8.

[118] David G. Kendall. “Some Problems and Methods in Statistical Archaeology”. In: *World Archaeology* 1.1 (1969), pp. 68–76. ISSN: 00438243, 14701375.

[119] Pascal Kerschke, Lars Kotthoff, Jakob Bossek, Holger H Hoos, and Heike Trautmann. “Leveraging TSP solver complementarity through machine learning”. In: *Evolutionary computation* 26.4 (2018), pp. 597–620.

[120] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd Int. Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings.* Ed. by Yoshua Bengio and Yann LeCun. 2015.

[121] Gözde Kizilateş and Fidan Nuriyeva. “On the Nearest Neighbor Algorithms for the Traveling Salesman Problem”. In: *Advances in Computational Science, Engineering and Information Technology.* Ed. by Dhinaharan Nagamalai, Ashok Kumar, and Annamalai Annamalai. Heidelberg: Springer International Publishing, 2013, pp. 111–118. ISBN: 978-3-319-00951-3.

[122] Lars Kotthoff. “Algorithm Selection for Combinatorial Search Problems: A Survey”. In: *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach.* Ed. by Christian Bessiere, Luc De Raedt, Lars Kotthoff, Siegfried Nijssen, Barry O’Sullivan, and Dino Pedreschi. Vol. 10101. LNCS. Springer, 2016, pp. 149–190. ISBN: 978-3-319-50136-9.

[123] Manolis Koubarakis. “The Complexity of Query Evaluation in Indefinite Temporal Constraint Databases”. In: *Theor. Comput. Sci.* 171.1-2 (1997), pp. 25–60.

[124] A. Krokhin, P. Jeavons, and P. Jonsson. “Reasoning about temporal relations: The tractable subalgebras of Allen’s interval algebra”. In: *Journal of the ACM* 50.5 (2003), pp. 591–640.

[125] P.B. Ladkin. “Models of Axioms for Time Intervals”. In: *Proc. of the 6th National Conference on Artificial Intelligence.* 1987, pp. 234–239.

[126] Gilbert Laporte, Hélène Mercure, and Yves Nobert. “Generalized travelling salesman problem through n sets of nodes: the asymmetrical case”. In: *Discrete Applied Mathematics* 18.2 (1987), pp. 185–197. ISSN: 0166-218X.

[127] Thierry Le Provost and Mark Wallace. “Generalized Constraint Propagation over the CLP Scheme”. In: *J. Log. Program.* 16.3 (1993), pp. 319–359.

[128] Christophe Lecoutre. “Optimization of Simple Tabular Reduction for Table Constraints”. In: *Principles and Practice of Constraint Programming, 14th International Conference, CP 2008, Sydney, Australia, September 14-18, 2008. Proceedings.* Ed. by Peter J. Stuckey. Vol. 5202. Lecture Notes in Computer Science. Springer, 2008, pp. 128–143.

[129] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. “Reasoning from last conflict(s) in constraint programming”. In: *Artif. Intell.* 173.18 (2009), pp. 1592–1614.

[130] Jan Karel Lenstra. “Clustering a data array and the traveling-salesman problem”. In: *Operations Research* 22.2 (1974), pp. 413–414.

[131] Paolo Liberatore. “On the complexity of choosing the branching literal in DPLL”. In: *Artificial Intelligence* 116.1 (2000), pp. 315–326. ISSN: 0004-3702.

[132] G. Ligozat. “A New Proof of Tractability for ORD-Horn Relations”. In: *AAAI-96 Proceedings*. 1996, pp. 395–401.

[133] Gerard Ligozat. “On Generalized Interval Calculi”. In: *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 1*. Ed. by Thomas L. Dean and Kathleen R. McKeown. AAAI Press / The MIT Press, 1991, pp. 234–240.

[134] Gérard Ligozat. “A New Proof of Tractability for 0RD-Horn Relations”. In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*. AAAI’96. Portland, Oregon: AAAI Press, 1996, 395–401. ISBN: 026251091X.

[135] Feng-Tse Lin, Cheng-Yan Kao, and Ching-Chi Hsu. “Applying the genetic approach to simulated annealing in solving some NP-hard problems”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 23.6 (1993), pp. 1752–1767.

[136] S. Lin and Brian W. Kernighan. “An Effective Heuristic Algorithm for the Traveling-Salesman Problem”. In: *Operations Research* 21.2 (1973), pp. 498–516.

[137] Michele Lombardi and Michela Milano. “Boosting Combinatorial Problem Modeling with Machine Learning”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018*. Ed. by Jérôme Lang. ijcai.org, 2018, pp. 5472–5478.

[138] Michele Lombardi, Michela Milano, and Andrea Bartolini. “Empirical decision model learning”. In: *Artif. Intell.* 244 (2017), pp. 343–367.

[139] A.K. Mackworth. “Consistency in Networks of Relations”. In: *Artificial Intelligence* 8.1 (1977), pp. 99–118.

[140] Alan K. Mackworth. “Consistency in networks of relations”. In: *Artificial Intelligence* 8.1 (1977), pp. 99–118. ISSN: 0004-3702.

[141] Oli BG Madsen. “An application of travelling-salesman routines to solve pattern-allocation problems in the glass industry”. In: *Journal of the Operational Research Society* 39.3 (1988), pp. 249–256.

[142] Vangelis F Magirou. “The efficient drilling of printed circuit boards”. In: *Interfaces* 16.4 (1986), pp. 13–23.

[143] M. Mantle, S. Batsakis, and G. Antoniou. “Large Scale Reasoning Using Allen’s Interval Algebra”. In: *Proc. of the 15th Mexican International Conference on Artificial Intelligence*. Vol. 11062. LNCS. 2017, pp. 29–41.

[144] William T McCormick Jr, Paul J Schweitzer, and Thomas W White. “Problem decomposition and data reorganization by a clustering technique”. In: *Operations Research* 20.5 (1972), pp. 993–1009.

[145] K. Menger. “Bericht über ein mathematisches Kolloquium”. In: *Monatshefte für Mathematik und Physik* 38 (1931), pp. 17–38.

[146] Olaf Mersmann, Bernd Bischl, Heike Trautmann, Markus Wagner, Jakob Bossek, and Frank Neumann. “A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem”. In: *Annals of Mathematics and Artificial Intelligence* 69.2 (2013), pp. 151–182.

[147] Steven Minton. “Automatically Configuring Constraint Satisfaction Programs: A Case Study”. In: *Constraints An Int. J.* 1.1/2 (1996), pp. 7–43.

[148] Roger Mohr and Gérald Masini. “Good Old Discrete Relaxation”. In: *Proceedings of the 8th European Conference on Artificial Intelligence.* ECAI’88. Munich, Germany: Pitman Publishing, Inc., 1988, 651–656. ISBN: 0273087983.

[149] Ugo Montanari. “Networks of constraints: Fundamental properties and applications to picture processing”. In: *Inf. Sci.* 7 (1974), pp. 95–132.

[150] Ugo Montanari. “Networks of constraints: Fundamental properties and applications to picture processing”. In: *Information Sciences* 7 (1974), pp. 95 –132.

[151] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Annual Design Automation Conference.* DAC ’01. Las Vegas, Nevada, USA: ACM, 2001, pp. 530–535. ISBN: 1-58113-297-2.

[152] L. Mudrová and N. Hawes. “Task scheduling for mobile robots using interval algebra”. In: *Proc. of the International Conference on Robotics and Automation.* 2015, pp. 383–388.

[153] B. Nebel. “Solving Hard Qualitative Temporal Reasoning Problems: Evaluating the Efficiency of Using the ORD-Horn Class”. In: *Constraints* 1.3 (1997), pp. 175–190.

[154] B. Nebel and H.J. Bürckert. “Reasoning about Temporal Relations: A Maximal Tractable Subclass of Allen’s Interval Algebra”. In: *Journal of the ACM* 42.1 (1995), pp. 43–66.

[155] Klaus Nökel and Hans Lamberti. “Temporally distributed symptoms in a diagnostic application”. In: *Artif. Intell. Eng.* 6.4 (1991), pp. 196–204.

[156] Christos H. Papadimitriou and Mihalis Yannakakis. “Scheduling interval-ordered tasks”. In: *SIAM Journal on Computing* 8.3 (1979), pp. 405–409.

[157] J. F. Pekny and D. L. Miller. “An Exact Parallel Algorithm for the Resource Constrained Traveling Salesman Problem with Application to Scheduling with an Aggregate Deadline”. In: *Proceedings of the 1990 ACM Annual Conference on Cooperation.* CSC ’90. Washington, D.C., USA: Association for Computing Machinery, 1990, 208–214. ISBN: 0897913485.

[158] Gilles Pesant, Michel Gendreau, Jean-Yves Potvin, and Jean-Marc Rousseau. “An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows”. In: *Transportation Science* 32.1 (1998), pp. 12–29.

[159] D.N. Pham, J. Thornton, and A. Sattar. “Modelling and solving temporal reasoning as propositional satisfiability”. In: *Artificial Intelligence* 172.15 (2008), pp. 1752–1782.

[160] Josef Pihera and Nysret Musliu. “Application of machine learning to algorithm selection for TSP”. In: *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*. IEEE. 2014, pp. 47–54.

[161] Patrick Prosser. “Hybrid Algorithms for the Constraint Satisfaction Problem”. In: *Comput. Intell.* 9 (1993), pp. 268–299.

[162] F. J. Provost and T. Fawcett. “Robust Classification for Imprecise Environments”. In: *Machine Learning* 42.3 (2001), pp. 203–231.

[163] Jean-François Puget. “PECOS: a high level constraint programming language”. In: *Proceedings of SPICIS*. Vol. 92. 1992.

[164] Jean-François Puget. “A C++ implementation of CLP”. In: *Proceedings of the Singapore Conference on Intelligent Systems (SPICIS’94, Singapore)*. 1994.

[165] Jean-Francois Puget. “A Fast Algorithm for the Bound Consistency of alldiff Constraints”. In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.* Ed. by Jack Mostow and Chuck Rich. AAAI Press / The MIT Press, 1998, pp. 359–366. ISBN: 0-262-51098-7.

[166] J. Ross Quinlan. “Induction of Decision Trees”. In: *Mach. Learn.* 1.1 (1986), pp. 81–106.

[167] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993. ISBN: 1-55860-238-0.

[168] M. Ragni and S. Wölfl. “Branching Allen”. In: *Proc. of the 4th International Conference on Spatial Cognition.* Vol. 3343. LNCS. 2004, pp. 323–343.

[169] H Donald Ratliff and Arnon S Rosenthal. “Order-picking in a rectangular warehouse: a solvable case of the traveling salesman problem”. In: *Operations research* 31.3 (1983), pp. 507–521.

[170] Jean-Charles Régin. “A Filtering Algorithm for Constraints of Difference in CSPs”. In: *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.* Ed. by Barbara Hayes-Roth and Richard E. Korf. AAAI Press / The MIT Press, 1994, pp. 362–367. ISBN: 0-262-61102-3.

[171] A.J. Reich. “Intervals, Points, and Branching Time”. In: *Proc. of TIME 1994: 9th International Symposium on Temporal Representation and Reasoning.* 1994, pp. 121–133.

[172] Gerhard Reinelt. “TSPLIB - A Traveling Salesman Problem Library”. In: *INFORMS Journal on Computing* 3.4 (1991), pp. 376–384.

[173] J. Renz and B. Nebel. “Efficient Methods for Qualitative Spatial Reasoning”. In: *Journal of Artificial Intelligence Resoning* 15 (2001), pp. 289–318.

[174] J. Renz and B. Nebel. “Qualitative Spatial Reasoning using Constraint Calculi”. In: *Handbook of Spatial Logic.* Springer, 2007, pp. 161–215.

[175] Jochen Renz and Ligozat Gérard. “Weak Composition for Qualitative Spatial and Temporal Reasoning”. In: Oct. 2005, pp. 534–548. ISBN: 978-3-540-29238-8.

[176] E. Rishes, S.M. Lukin, D.K. Elson, and M.A. Walker. “Generating Different Story Tellings from Semantic Representations of Narrative”. In: *Proc. of the 6th International Conference on Interactive Storytelling.* Vol. 8230. LNCS. 2013, pp. 192–204.

[177] J. Robinson. “On the Hamiltonian Game (a Traveling Salesman Problem)”. In: *RAND Research Memorandum RM-303.* RAND Corporation, 1949.

[178] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming.* Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006. ISBN: 978-0-444-52726-4.

[179] G. Rosu and S. Bensalem. “Allen Linear (Interval) Temporal Logic - Translation to LTL and Monitor Synthesis”. In: *Proc. of the 18th International Conference on Computer Aided Verification.* Vol. 4144. LNCS. 2006, pp. 263–277.

[180] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-Propagating Errors”. In: *Neurocomputing: Foundations of Research.* Cambridge, MA, USA: MIT Press, 1988, 696–699. ISBN: 0262010976.

[181] Hanan Samet. “The Quadtree and Related Hierarchical Data Structures”. In: *ACM Comput. Surv.* 16.2 (1984), pp. 187–260.

[182] Robert J Sault, Peter J Teuben, and Mel CH Wright. “A retrospective view of MIRIAD”. In: *Astronomical Data Analysis Software and Systems IV.* Vol. 77. 1995, p. 433.

[183] Thomas Schiex and Gérard Verfaillie. “Nogood Recording for Static and Dynamic Constraint Satisfaction Problems”. In: *Int. J. Artif. Intell. Tools* 3.2 (1994), pp. 187–208.

[184] Joachim Schimpf and Kish Shen. “ECL^{i}PS^e - From LP to CLP”. In: *TPLP* 12.1-2 (2012), pp. 127–156.

[185] Michael Sioutis, Anastasia Paparrizou, and Tomi Janhunen. “On neighbourhood singleton-style consistencies for qualitative spatial and temporal reasoning”. In: *Information and Computation* (2020), pp. 1–17. ISSN: 0890-5401.

[186] Kate Smith-Miles, Jano I. van Hemert, and Xin Yu Lim. “Understanding TSP Difficulty by Learning from Evolved Instances”. In: *Learning and Intelligent Optimization, 4th International Conference, LION 4, Venice, Italy, January 18-22, 2010. Selected Papers.* Ed. by Christian Blum and Roberto Battiti. Vol. 6073. LNCS. Springer, 2010, pp. 266–280.

[187] Fei Song and Robin Cohen. “The Interpretation of Temporal Relations in Narrative”. In: *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988.* Ed. by Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith. AAAI Press / The MIT Press, 1988, pp. 745–750.

[188] Richard M. Stallman and Gerald J. Sussman. “Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis”. In: *Artificial Intelligence* 9.2 (1977), pp. 135–196. ISSN: 0004-3702.

[189] M. Theune, K. Meijs, D. Heylen, and R.Ordelman. “Generating expressive speech for storytelling applications”. In: *IEEE Transactions on Audio, Speech & Language Processing* 14.4 (2006), pp. 1137–1144.

[190] Edward P. K. Tsang. *Foundations of constraint satisfaction.* Computation in cognitive science. Academic Press, 1993. ISBN: 978-0-12-701610-8.

[191] P. van Beek and R. Cohen. "Exact and approximate reasoning about temporal relations". In: *Computational Intelligence* 6 (1990), pp. 132–144.

[192] M. B. Vilain and H.A. Kautz. "Constraint Propagation Algorithms for Temporal Reasoning". In: *Proc. of the 5th National Conference on Artificial Intelligence.* 1986, pp. 377–382.

[193] Stephen A. Ward and Robert H. Halstead Jr. *Computation structures.* MIT electrical engineering and computer science series. MIT Press, 1990. ISBN: 978-0-262-23139-8.

[194] M. Westphal, J. Hué, and S. Wölfl. "On the Propagation Strength of SAT Encodings for Qualitative Temporal Reasoning". In: *Proc. of the 25th International Conference on Tools with Artificial Intelligence.* 2013, pp. 46–54.

[195] M. Westphal and S. Wölfl. "Qualitative CSP, Finite CSP, and SAT: Comparing Methods for Qualitative Constraint-based Reasoning". In: *Proc. of the 21st International Joint Conference on Artificial Intelligence.* 2009, pp. 628–633.

[196] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* 3rd ed. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2011. ISBN: 978-0-12-374856-0.

[197] A.K. Zaidi and L.W. Wagenhals. "Planning temporal events using point-interval logic". In: *Mathematical and Computer Modelling* 43.9 (2006), pp. 1229 –1253.

[198] Neng-Fa Zhou. "Encoding Table Constraints in CLP(FD) Based on Pair-Wise AC". In: *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings.* Ed. by Patricia M. Hill and David Scott Warren. Vol. 5649. Lecture Notes in Computer Science. Springer, 2009, pp. 402–416. ISBN: 978-3-642-02845-8.