Robinson Andrey Duque

Applying Learning Techniques in SABIO, a Constraint Programming Application for Soccer Analysis

Tesis de Maestría en Ingeniería
con Énfasis en Ingeniería de Sistemas y Computación

Universidad del Valle
February - 2016

Facultad de Ingeniería
Escuela de Ingeniería de Sistemas y Computación

Applying Learning Techniques in SABIO, a Constraint Programming Application for Soccer Analysis

Tesis de Maestría en Ingeniería
con Énfasis en Ingeniería de Sistemas y Computación
by

Eng. Robinson Andrey Duque
robinson.duque@correounivalle.edu.co

Supervised by

Juan Francisco Díaz, Ph.D., Universidad del Valle
juanfco.diaz@correounivalle.edu.co
Alejandro Arbelaez, Ph.D., University College Cork
a.arbelaez@ucc.ie

Santiago de Cali
February - 2016

To my beloved parents, who taught me that
a life project can only be achieved through
dedication, persistence and determination.

Acknowledgments

Firstly, I would like to express my sincere gratitude to my director Dr. Juan Francisco Díaz who gave me the opportunity to join the Avispa research group and also for his patience, motivation and continuous support in my master studies and related research[1]. Without his support and his academical guidance, it would've not been possible to complete this research.

My sincere gratitude also goes to Dr. Alejandro Arbelaez for his support during the course of this research and also for sharing with me his wide knowledge on the field of study that let me explore this research from various perspectives.

SABIO is a platform created with the academic support of the AVISPA research group from the Universidad del Valle. I would like to thank Luis Felipe Vargas, María Andrea Cruz and Carlos Martínez for working in early versions of the development of SABIO under the supervision of Dr. Juan Francisco Díaz.

Last but not the least, I would like to thank my parents Yadira Agudelo and Gerardo Duque for their unconditional support and also for educating me through the example of working hard to achieve all my goals.

[1] Robinson Duque is supported by the Colombian Administrative Department of Science, Technology and Innovation(Colciencias) under the PhD scholarship program.

Abstract

Soccer is one of the most popular sports in the world with a big impact in mass media and followed by millions of people. A soccer competition consists of n teams playing against each other according to a single or double round-robin schedule and it can be played as a league or tournament system. These competitions offer an excellent opportunity to model interesting problems related to questions that soccer fans frequently ask about their favourite teams. For instance, at some stage of the competition, fans might be interested in determining whether a given team still has chances of winning the competition (i.e., finishing first in a league or being within the first k teams in a tournament to qualify to the playoff). This problem relates to the elimination problem, which is NP-complete for the actual FIFA pointing rule system (0, 1, 3), zero point to a loss, one point to a tie, and three points to a win.

Constraint programming (i.e., CP) is a paradigm that has developed several techniques to address combinatorial problems and in the past years, a growing interest in combining CP and Machine learning has been studied in order to improve CP efficiency when searching for feasible solutions. SABIO stands for *Soccer Analysis Based on Inference Outputs* and it is a CP platform created with the academic support of the AVISPA research group from the Universidad del Valle (`www.sabiofutbol.com`). SABIO can be used to discover information and answer questions related to soccer teams from a particular soccer competition and offers graphical interfaces to set up questions in form of constraints. To find solutions, SABIO uses branching strategies that includes heuristics with static variable and value selection that works well for certain instances. However, empirical tests managed to identify a family of instances where SABIO presented low performance and the running times to find feasible answers were too long (more than 30 seconds) to be offered in a web or mobile application.

To address this shortcoming, we combined the use of constraint programming, machine learning, and mixed integer programming to propose 2 models to solve general soccer fan queries at different stages of a competition. The first model (i.e., CP-ML) combines a series of new constraints to increase pruning and a machine learning classifier for value selection. The second model (i.e., MIP) is a mixed integer programming version of SABIO. Our computational experiments showed that this new models improve the performance of SABIO if used independently. Additionally, we found that combining both models in a mixed execution, become a very robust and fast solver.

Resumen

El fútbol es uno de los deportes más importantes en el mundo, con un gran impacto en los medios masivos de comunicación y seguido por millones de personas. Una competencia de fútbol consiste en n equipos jugando en un sistema de todos contra todos, ya sea en un torneo o liga. Estas competencias ofrecen una excelente oportunidad para modelar problemas relacionados con preguntas que los hinchas frecuentemente tienen sobre sus equipos favoritos. Por ejemplo, en determinado momento, los hinchas pueden estar interesados en determinar si un equipo aún tiene posibilidades de ganar la competencia, es decir, quedar de primero en la liga o finalizar dentro de los primeros k equipos en un tornero para clasificar al sistema de eliminación. Esta pregunta se denomina el problema de la eliminación, el cual es NP-Completo para el sistema actual de puntos determinado por la FIFA (0, 1, 3). Cero puntos cuando se pierde, un punto cuando se empata, tres puntos cuando se gana.

La programación por restricciones (CP), es un paradigma que ha desarrollado diversas técnicas para abordar problemas combinatorios y en los últimos años, un creciente interés en combinar CP con aprendizaje de máquina (ML) ha sido estudiado con el fin de mejorar la eficiencia de CP en la búsqueda de soluciones. SABIO es acrónimo de *Soccer Analysis Based on Inference Outputs*, una plataforma desarrollada bajo el paradigma de CP con el soporte académico de AVISPA, un grupo de investigación de la Universidad del Valle (`www.sabiofutbol.com`). SABIO puede ser usado para descubrir información y responder preguntas relacionadas con equipos de fútbol en una competencia particular. La plataforma ofrece interfaces gráficas para establecer preguntas en forma de restricciones. Para encontrar soluciones, SABIO utiliza estrategias de búsqueda que incluyen selección estática de variable/valor que funcionan bien para ciertas instancias. Sin embargo, pruebas empíricas sobre la aplicación, permitieron identificar una familia de instancias donde SABIO presenta bajo rendimiento para encontrar soluciones y los tiempos de ejecución son muy largos (más de 30 segundos) para ser ofrecidos en una aplicación web o móvil.

Para afrontar esta dificultad, hemos combinado el uso de programación por restricciones, aprendizaje de máquina y programación entera mixta (MIP) para proponer 2 modelos que resuelvan preguntas relacionadas con fútbol en cualquier momento de la competencia. El primer modelo (CP-ML) combina una serie de nuevas restricciones que incrementan la poda en el árbol de búsqueda, además de un clasificador para selección de valores. El segundo modelo (MIP) es una versión de programación entera mixta de SABIO. Nuestros experimentos demuestran que estos nuevos modelos mejoran el rendimiento de SABIO. Encontramos también que si ambos modelos se combinan en una ejecución mixta, se convierten en un solver muy robusto y rápido.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Constraint programming (CP) is a paradigm for solving combinatorial search problems that includes techniques from artificial intelligence, computer science and operations research that deal with reasoning and computing [1]. CP has been applied successfully to find solutions to many combinatorial problems in real-life areas like circuit verification, resource allocation, planing, scheduling, circuit design, DNA sequencing, real time control systems, etc [2]. CP generally consists of a two phases approach: first, a representation or model of the problem is created by means of constraints and then the problem is solved by using constraint satisfaction techniques and a search strategy [3]. *The model* or CSP *(Constraint satisfaction problem)* consists of a finite set of constraints, each on a given sequence of variables [3] and its *solution* can be reached by a search method that specifies heuristics to find one or more solutions.

CP uses reasoning techniques like constraint propagation that combined with a search heuristic can greatly reduce the search space. Algorithms that implement heuristics to solve a CSP can perform well on different problem instances, but it's also known that there is rarely a best algorithm for a given problem, because the heuristics can fail and perform poorly in some other instances. This phenomenon is common in algorithms that attempt to find solutions in NP problems, because runtimes are often highly variable from instance to instance [4]. Nevertheless, in the past two decades machine learning and constraint programming have been studied widely by computer science researchers to mitigate this problem, including different perspectives, such as *portfolio algorithms*, *per-class learning* and *adaptive*

algorithms [5] to include learning in the search process.

SABIO (Soccer Analysis Based on Inference Outputs) is a CP application created by the AVISPA research group from the Universidad del Valle that offers graphical interfaces to the end users in order to set up soccer queries in form of constraints and help users discover information related to soccer teams from a particular competition. To find solutions, SABIO uses a branching strategy that includes heuristics with static variable and value selection that work well for certain instances, but there are queries for which the implemented heuristics in the search algorithm fail and the running times to find answers are too long to be offered in a web or mobile application.

1.1. Problem Approach

Real world constraint problems usually require to explore large combinatorial spaces which can be computationally expensive. Most of these combinatorial problems require considerable experimentation, complex algorithms and data structures to offer reasonable solutions [6], therefore, it is very unlikely that efficient general-purpose algorithms exist for solving all the problem instances, however practical applications arise instances that have special forms that enable them to be solved more efficiently [1].

At the beginning of CP many *static search strategies* were considered, but better results are obtained when *dynamic ordering strategies* are introduced [7]. These strategies compute the next variable to assign during search by using a criteria measurement and the variable having the best value for this criteria is then chosen. There is a growing interest in the research community to study alternative heuristics that include *learning techniques* for selecting the *variable order* in which they should be instantiated and heuristics for*selecting the values* to use during the search process, in order to improve performance. The field of Machine learning offers a promising approach to improve CP performance by implementing different strategies such as portfolio algorithms, per-class learning, adaptive algorithms, constraint learning and so on.

Soccer or football is one of the most popular sports in the world with millions of fans. A soccer competition consists of n teams playing against each other according to a single or double round-robin schedule and it can be played as a league or tournament system. These competitions offer an excellent opportunity to model interesting problems related to questions that soccer fans usually ask about their favourite teams. For instance, at some stage of the competition, fans might be interested in determining if team i still has a theoretical chance to win the competition (i.e., finish first in a league or be within the first k teams in a tournament to qualify for playoffs). This problem relates to the *elimination elimination problem* and turns out to be NP-Complete for the current FIFA score system (0, 1, 3), zero point to a loss, one point to a tie, and three points to a

win [8], [9].

Common soccer computational problems (e.g., the elimination problem, possible qualification problem, the score vector problem, etc.) have been studied before using linear programming (see section 3 for further information). However, to the best of our knowledge, SABIO is the first interactive platform that can be used to represent several soccer computational problems using a constraint programming approach. It offers a general model that can be used to simulate different scenarios and problems where fans can combine four different kind of constrains to set up queries to analyze soccer:

- *Game Results.* These kind of queries let users impose constraints about particular matches in any fixture. (e.g., Barcelona ends in a tie with R.Madrid).
- *Position in Ranking.* These kind of queries let users impose constraints about the positions of the teams at the end of a tournament (e.g., R.Madrid will be in a better position than 3).
- *Relative Position.* These kind of queries let users impose constraints about the positions of two teams (e.g., R.Madrid will be in a worse position than Barcelona).
- *Final Points.* These kind of queries let users impose constraints about the expected final points of the teams at the end of a tournament (e.g., Barcelona score at the end of the tournament is 75).

SABIO implements a static branching strategy that orders the variables per team and define their search strategy before starting the actual search. It has been observed that these heuristics for variable/value selection sometimes fail and don't have the expected effectiveness and efficiency to offer a service in a web or mobile application due to the long response times. On the other hand, SABIO has a module to configure the search algorithm, and it can be seen as an attempt solve the time consuming queries by directing the search to more feasible sections of the search space, nevertheless, it requires that the user understands how the branching takes place and most of the potential users of SABIO might not have such kind of understanding. In the following numerals we will attempt to describe the main goals of this dissertation that established the scope and the contributions of this research.

1.2. Addressed Goals

Previously we described briefly that SABIO is the context in which we are carrying out this research. In this way, as part of the main goal of this dissertation we attempt to:

Implement learning techniques in the search algorithm of SABIO to improve its capacity to answer a wider range of instances by studying aspects related to machine learning in constraint programming.

This goal was set at the beginning of the research, however it is a general goal and it was split in four specific goals that helped us determine the scope and keep track of the progress of the research. Namely, the four specific goals at the beginning of the research were:

- *G1: To create a state of the art related to machine learning in constraint programming.*
- *G2: To identify family of instances where SABIO presents low performance and that can be used to train and validate the model.*
- *G3: To determine the most appropriate learning techniques to improve the search algorithm of SABIO according to the actual characteristics of the application.*
- *G4: To determine the features and instances that will be used to train and validate the model by using performance tests.*

1.3. Contribution Overview

In order to complete goal *G1* we (i) provide a summary of the most relevant concepts of constraint programming, machine learning and integer programming that are related to this dissertation. Furthermore, during the research we also found convenient to determine the kind of problems that can be tackled with our model, therefore, we decided to (ii) provide a state of the art related to common soccer computational problems, in order to establish a reference point of previous approaches that can also be addressed with SABIO. To accomplish goal *G2*, we (iii) performed a series of tests to the former CP version of SABIO and managed to identify a family of instances where SABIO presented low performance. This family of instances was used to fulfill goals *G3* and *G4*, for which we (iv) proposed an improvement to the former CP version by using a classifier that extends the branching heuristics and we also proposed a series of constraints in order improve pruning during search to answer a wider range of instances within acceptable times to be offered in a web application. Finally, during this research we found out that previous soccer problems were tackled using integer programming, therefore, we decided to (v) implement a MIP (mixed integer programming) version of SABIO using CPLEX and compare the performance of the new CP model and the MIP model. Finally, we (vi) proposed a sequential execution of both models to improve SABIO's efficiency and effectiveness.

1.4. Thesis Organization

The remainder of this document is structured as follows:

- **Chapter 2 (Preliminaries).** Includes an overview of the most relevant concepts to this dissertation like *constraint programming* and the related notions of propagation, search, and variable/value selection heuristics. We also include an overview of *machine learning* in the context of constraint satisfaction problems and a slight review of *integer programming* given that a MIP version of SABIO is proposed in Chapter 6.
- **Chapter 3 (Soccer computational problems).** Introduces previously studied soccer problems that can be addressed with the computational model of SABIO; e.g., the elimination and the score vector problems. It also contains a review of some previous integer programming approaches that can also be represented with SABIO.
- **Chapter 4 (SABIO as a CP application for soccer analysis).** Contains a detailed explanation of the former CP model of SABIO and we also present a series of tests that we used to identify and analyse a family of hard instances for SABIO.
- **Chapter 5 (SABIO - CSP improvements and machine learning).** In this chapter, we introduce an improvement to the former CP model by training a classifier for value selection and implementing some pruning rules that improve the search performance.
- **Chapter 6 (SABIO as a MIP application for soccer analysis).** We present a Mixed Integer Programming version of SABIO in order to tackle soccer problems. This chapter also contains empirical test and results of comparing the models proposed in this document.
- **Chapter 7 (Conclusions and future work).** We present our conclusions and we also describe the research lines that can be further explored on the basis of this work.

References

[1] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[2] M. Wallace, «Practical applications of constraint programming», *Constraints*, vol. 1, no. 1-2, pp. 139–168, 1996.

[3] K. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[4] K. Leyton Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, «A portfolio approach to algorithm selection», in *IJCAI*, vol. 1543, 2003, p. 2003.

[5] A. Arbelaez, «Learning during search», PhD thesis, Université Paris, 2011.

[6] L. Michel and P. V. Hentenryck, «A constraint-based architecture for local search», in *ACM SIGPLAN Notices*, ACM, vol. 37, 2002, pp. 83–100.

[7] J.-C. Régin, «Solving problems with cp: Four common pitfalls to avoid», in *Principles and Practice of Constraint Programming–CP 2011*, Springer, 2011, pp. 3–11.

[8] T. Bernholt, A. Gälich, T. Hofmeister, and N. Schmitt, «Football elimination is hard to decide under the 3-point-rule», in *MFCS*, 1999, pp. 410–418.

[9] W. Kern and D. Paulusma, «The new fifa rules are hard: Complexity aspects of sports competitions», *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 317–323, 2001.

Chapter 2

Preliminaries

Contents

2.1. Introduction

Combinatorial problems and optimization are usually of interest for academic and industry research and it frequently involves fields from machine learning, artificial intelligence, mathematics, and software engineering. Mathematical and constraint programming have been studied widely over several years and have brought new hybrid approaches that combined with machine learning can be used to solve complex planning and scheduling problems.

One of the goals of this research (i.e., *G1* depicted in section 1.2) includes a review of machine learning in the context of constraint programming, therefore, this chapter presents in section 2.2 a brief review the most important concepts of Constraint Programming (CP) like propagation and search and also includes a short review of the most common variable/value selection heuristics. In section 2.3

we present an overview of the most common approaches (e.g., algorithm portfolios, per-class learning) that combine the usage of machine learning in constraint satisfaction problems. Finally, in section 2.4, we briefly present the main concepts of Mixed Integer Programming (MIP) given that a MIP version of SABIO is proposed in Chapter 6.

2.2. Constraint Programming

2.2.1. Basic Concepts

Constraint programming is a powerful paradigm that consists of a group of techniques that can be used to solve combinatorial problems. It has been applied successfully in many real-life areas like circuit verification, real time control systems, resource allocation, planing and scheduling circuit design, DNA sequencing and real time control systems[2].

A *Constraint Satisfaction Problem* or CSP, is a triplet $\langle X, D, C\rangle$, where X is a set of variables, D the domain of the variables and C is a set of constraints [10]. Formally, Apt defines a CSP as a finite sequence of variables $X := x_1, ..., x_m$ with respective domains $D_1, ..., D_n$, with a finite set of constraints C, each on a subsequence of variables of X [3]. A CSP can be written as:

$$\langle C; x_1 \in D_1, ..., x_n \in D_n\rangle$$

A solution to a CSP is an assignment of legal values to each variable, such that all its constraints are satisfied. Each solution $(d_1, ..., d_n)$ corresponds to a unique solved CSP:

$$\langle C'; x_1 \in \{d_1\}, ..., x_n \in \{d_n\}\rangle$$

In a general description, constraint programming consists of two phases: $CP = Model + Search$ [11]. First, a model or CSP of the problem is created and then the problem is solved by using constraint satisfaction techniques and a search strategy [3]. When solving a CSP, the programmer can be interested in determine if the CSP has a solution (The CSP is Consistent), find one or all the solutions and find an optimal solution using some kind of quality measure.

Constraint programming enables the users to define global constraints to represent complex relations between multiple variables. Such global constraints can handle lists of variables whose length doesn't need to be fixed, for instance the *alldifferent* constraint which is usually encoded into the CP constraint system [12].

The main advantage of CP systems is that users can also define new problem specific constraints by using a richer range of operators than those used in mathematical modelling. For instance, CP systems support all the usual mathematical constraints $LHS \geq RHS, LHS \leq RHS, LHS = RHS$ and some additional constraints like $LHS > RHS, LHS < RHS, LHS \neq RHS$. CP can also admit polynomials, trigonometric functions, max, min, abs and many other functions

which extends their expressive power unlike integer programming and its linear expressions [12].

A CSP can be solved by brute search, enumerating all the values of the variables and checking all combinations for possible solutions, but the number of combinations are usually too large to check them all [13]. CSPs are well known to be NP-Complete and the research community has focused in studying alternative heuristics that include *learning techniques* for *selecting the variable order* in which they should be instantiated and heuristics for *choosing which value* to use during the search process.

2.2.2. Propagation in CP

Once the CSP model has been formulated, it can be solved using propagation and search algorithms. The possible combinations of the variable values of the CSP, generate a search space that can be seen as a search tree. The traditional approach to solve a constraint satisfaction problem is based on *depth-first search* with backtracking combined with propagation which requires polynomial-time inference at each node in the search tree to reduce the amount of search needed [14].

Propagation is a procedure that replaces a given CSP by a "simpler" one, but equivalent [3]. The general idea of propagation is to reduce the constraints or the domain size by removing probably inconsistent values of the variables, thus generating smaller spaces to search for solutions. These algorithms achieve different forms of *local consistency* and the domain reduction is performed repeatedly while maintaining equivalence. Let's suppose the following CSP:

$$\langle X + Y =: 1; X\,\{0,1,2,3,4\}, Y\,\{0,1,2,3,4\}\rangle$$

A propagation algorithm determines that the domains of the variables can be reduced to: $X\,\{0,1\}, Y\,\{0,1\}$, because there are no values in the domain of Y that satisfy the constraint $X + Y =: 1$ when X is $\{2,3,4\}$, neither values in the domain of X that satisfy the constraint when Y is $\{2,3,4\}$. In this case the CSP can be reduced to an equivalent CSP with smaller domains:

$$\langle X + Y =: 1; X\,\{0,1\}, Y\,\{0,1\}\rangle$$

If after propagation, a variable domain becomes empty, then a failure occurs. Propagation algorithms are usually incomplete, for instance, in some occasions, propagation is enough to find the solution of the CSP; that is, after applying propagation procedures, the variables might have a single value in their domains; nevertheless, like in the previous example, when variables have more than one possible value, solving the CSP involves a search algorithm to find a solution or an assignment of values to variables such as all constraints are satisfied.

CP propagation handles each constraint independently from the others and uses mathematical constraints to reason about the upper and lower bounds of

the variables occurring in it until reaching a fix-point (i.e., the constraints cannot propagate any further) [12]. In order to maintain the equivalence of the problem while changing the variable domains, constraint propagation algorithms achieve different forms of local consistency: *node consistency* for unary constraints, *arc consistency* for binary constraints and *hyper-arc-consistency* or often called *generalized arc consistency* (GAC) for n-ary constraints [3].

2.2.3. Search in CP

When propagation can not reach more reductions, a search algorithm (also called *distribution algorithm*) has to be used to solve the CSP. Two of the most known techniques are backtracking for a depth-first search and branch and bound when searching for an optimal solution[3].

The general approach according to [13] when no more local deductions can be done is to do a small search step and then do local deduction again. A search step splits the CSP P into two new problems $(P \wedge C)$ and $(P \wedge \neg C)$. Since C is a new constraint, each problem can do local deductions and the solutions of the CSP P is the union of the solutions of the two problems. The choice of C is extremely important and in some CP languages, the search can be quite sophisticated and can specify variable and value selection heuristics that could lead to a solution in just a few search steps. Newcomers discovering CP often overlook the true potential of search specification and fail to exploit it[11].

There are two well established techniques for solving CSPs: *complete techniques*, developed on top of a backtracking algorithm which combines a tree-based search with constraint propagation and *incomplete techniques*, based on local search algorithms to quickly find an assignment for each variable that satisfies all constraints[5].

One of the main drawbacks of local search is that it can reach a local minimum very quickly, therefore, many techniques have been proposed in order to overcome this situation. Some of the techniques include data structures to prevent cycling or visiting previous observed states, some other include random-walks and also restarts from new configurations after a given number of iterations. All these techniques specify variable and value selection heuristics and apart from *depth-first search*, there are many sophisticated algorithms like best-first search, large neighborhood search [15], limited discrepancy search [16], restarting strategies and metaheuristics for local search like ant colony optimization [17], tabu search [18], [19], simulated annealing [20], [21] just to name a few. This flexibility is mostly absent in mathematical programming where the search can be seen as a *black-box* that is controlled through a collection of parameters.

2.2.4. Variable/Value Selection Heuristics

The order in which variables are assigned by a backtracking search algorithm can lead to different results in terms of efficiency. Some of the most common value selection strategies in the literature are: *min-value* which selects the minimum value, *mid-value* selects the median value, *max-value* used to select the maximum value and *random-value* which selects a random value from the remaining domain of a variable [5]. Many search strategies have been studied and there are tools that implement several variable and value selection algorithms, for instance:

- *Naive:* Given a vector of variables Xv. Considers only the non-determined variables of Xv. Chooses the leftmost variable X in Xv. Creates a choice point for $X = L$ and $X \neq L$, where L is the lower bound of the domain of X [22].
- *ff(First Fail):* First fail is the oldest and probably the most common general purpose heuristic for variable/value selection, "To succeed, try first where you are most likely to fail". Given a vector of variables Xv. Considers only the non-determined variables of Xv. Chooses the leftmost variable X in Xv with the smallest domain size. Creates a choice point for $X = L$ and $X \neq L$, where L is the lower bound of the domain of X [22].
- *Split:* Given a vector of variables Xv. Considers only the non-determined variables of Xv. Chooses the leftmost variable X in Xv with the smallest domain size. Creates a choice point for $X \leq M$ and $X > M$, where M is the middle of the domain of X [22].

These variable/value strategies can work properly in some CSPs, nevertheless, several approaches have been proposed because better results can be obtained if the problem is analysed and a search strategy is created for each particular problem and there are mainly two categories of variable ordering heuristics:

- *Static (or fixed) variable ordering heuristics:* This type of heuristics only exploit the problem information in the initial state and maintain the same ordering during the search. Some examples of this kind of heuristics are *lexico*, which selects the variables lexicographically (i.e., the first unassigned variable in the list of decision variables), also *deg* and *ddeg* that order the variables decreasingly according to their initial degree [23].
- *Dynamic variable ordering heuristics:* This type of heuristics take into account some information during the search. A common dynamic heuristic is *dom* where variables are increasingly ordered according to the current size of their domains based on the first-fail principle [24]. There are also some heuristics that combine domain sizes with variable degrees like the case of *dom/deg* [25].

No heuristic outperforms the others but dynamic heuristics are usually more effective. In recent years, many improvements to backtracking that concern to ordering heuristics, filtering techniques, and conflict analysis have been proposed. Such improvements can be classified as look-ahead and look-back schemes where Look-ahead is used to extend the current partial solution by maintaining some level of local consistency, in order to anticipate future conflicts. On the other hand, Look-back schemes are designed to learn from past conflicts in order to avoid the same conflict later in the search. Some approaches include a combination of both schemes like the one proposed in [26], a dynamic and adaptive variable ordering heuristic *wdeg* and *dom/wdeg* that is able to direct systematic search to inconsistent or hard parts of a CSP, inspired by SAT solvers that use conflicts to drive a variable ordering heuristic by assigning higher weights to constraints violated at some previous states of the search process. *wdeg* selects the variable that is involved in most failed constraints based on a mechanism that associates a weight to each constraint that is incremented every time the constraint fails while *dom/wdeg*, combines the current domain of the variable and its degree in order to maximize the given ratio.

2.3. Machine Learning in Constraint Satisfaction Problems

Learning in the context of Constraint Solving is a technique by which unknown constraints are found during search and used to speed up subsequent search[27]. Machine learning is inherently a multidisciplinary field that is concerned with the question of how to construct computer programs that automatically improve with experience. Many practical computer programs that implement the use of types of learning have been developed successfully to recognize spoken words, detect fraudulent use of credit cards, drive autonomous vehicles, play games, etc [28].

The range of problems where machine learning can be applied is very large and researchers have identified a growing number of situations where it can be used, including learning associations, binary classification, multiclass classification, regression, clustering, reinforcement learning, etc [29][30].

In the past two decades machine learning and constraint programming have been studied widely by computer science researchers. This growing interest in both fields have brought awareness that machine learning can be used in constraint programming to improve performance. In 2011, the seminar *Constraint Programming meets Machine Learning and Data Mining*[31], brought together researchers from both communities to discuss how these fields could benefit from each other. Two of the discussion topics of the seminar included in the report [31] were:

- *Learning the model of the problem* by the use of machine learning and data

mining to learn appropriate models from examples and improve constraint programming. The conclusions on this topic were that it's possible to learn the triplet (V, D, C), with V the variables, D the domain of the variables and C the constraints.

- *Learning to solve* by the use of machine learning and data mining to improve the efficiency and quality of constraint solving using experience. In this case, the model (V, D, C) is given and passed runs of the solver are used to learn additional constraints or heuristics using inductive or deductive techniques. Inductive techniques would gather data from previous test-runs and learn constraints or heuristics; the learned hypotheses have to be tested against the model rather than against the data. Deductive techniques would employ methods of explanation based learning to analyse traces and proofs and deductively obtain constraints that are implied by the given model.

Machine learning offers many opportunities to solve specific problem instances and also has been applied in a range of problems that involve: *portfolio of constraint solvers*, to select the most appropriate algorithm for a given problem instance; *per-class learning*, to select the most appropriate parameter configuration for a given algorithm on a given class of problems; *adaptive algorithms*, to internally adapt the parameters of a given algorithm based on problem changing conditions [5].

Artificial intelligence has focused on developing new algorithms in order to solve similar instances for several years. In some scenarios, the new algorithm is superior to previous ones, however, it can not offer good results for certain instances, because it might have heuristics that are not satisfied in some cases. It's unlikely that a single algorithm works properly for every instance of a problem. A form to mitigate this situation in to use *algorithm portfolios*, which is a set of algorithms from which it's possible to select one with the best performance, depending on the problem instance [32].

The term "algorithm portfolio" was first introduced by Huberman, Lukose, and Hogg in 1997 [33] where they describe a strategy to execute several algorithms in parallel. However, back in 1975 Rice proposed a model [34] for algorithm selection given a space of instances and algorithms, every instance-algorithm was mapped to obtain its performance and then this mapping was used to train a model to select the best algorithm given a particular instance. Such a model is mainly a function $p(A, x)$ which maps an instance x of a problem space P into an algorithm $A_i \in A_1, ..., A_n$, where A_i is the most suitable algorithm to solve x, based on some performance criteria (See figure 2.1). This *winner take all* approach can ignore algorithms that might be competitive, so other proposals that combine algorithms in a portfolio were introduced by Gomes and Selman [35] in 1997 and machine learning was established as a method of addressing this kind of problem.

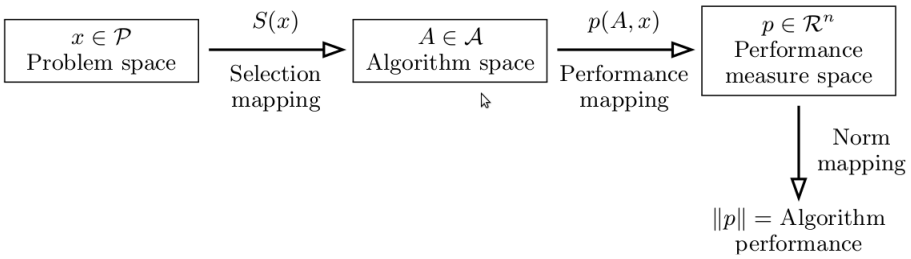

Figure 2.1: Rice's basic model for the algorithm selection problem [34].

Traditional models that offer an algorithm prediction over a particular problem, involve a *training phase* with the candidate algorithms $A_i \in A$ executing on a problem space $x \in P$ in order to evaluate their performance on a set of features or problem descriptors. These data is then used to create a performance model that can be used to predict performance for unseen problems:

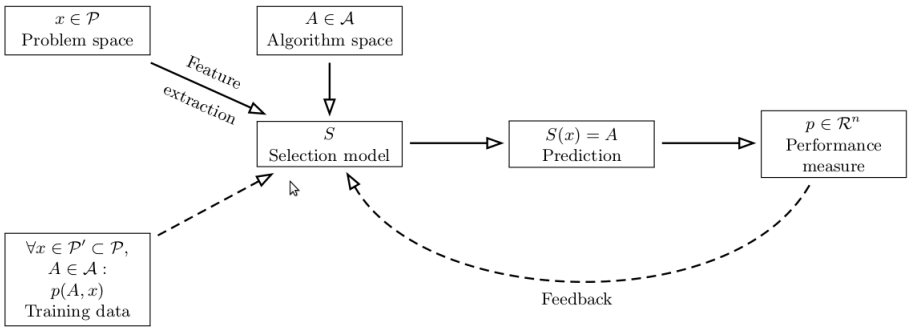

Figure 2.2: Basic framework using machine learning to train an algorithm selector.

One of the most outstanding work in the context of CSPs and algorithm portfolios is the CPhydra solver that was propose in [36] and uses (CBR) *Case Base Reasoning*, a lazy learning approach that contains a set of past examples called *cases* that can be used to determine how to solve an unknown instance, taking advantage of learned experience. CPhydra maintains a database with all solved instances or cases, then when a new instance I arrives, a set of similar cases C is computed and analysed in order to determine a set of solvers that maximizes the possibilities to solve the given instance, rather than a single solver.

The portfolio methodology has also been successfully applied in SATzilla to tackle SAT problems [37]. SATzilla uses machine learning to build an *empirical hardness model* to predict the time that an algorithm takes to solve a problem by using linear regression. SATzilla is one of the most important systems that implements portfolios for algorithm selection and it was the first system implementing portfolios to dominate the international SAT competition in 2007 [32]. This model could be seen as an oracle that allows to foresee the execution time, given certain features of a problem instance.

Another approach that uses algorithm portfolios is AQME. This system is a portfolio built on the Weka data mining library to solve QBFs Quantified Boolean Formulas, i.e. SAT instances with some universally quantified variables [38]. Some of the implementations of AQME have used decision trees, logistic regression, 1 nearest neighbour and decision rules.

A metaheuristic is a generic algorithmic template that needs to be configured or tuned, in order to find high quality solutions of hard combinatorial optimization problems. In general, the tuning problem rests on the concept of *class of instances* and consists in finding the best possible configuration of an algorithm, that yields the best results on the instances that the algorithm will attempt to solve [39].

Most of the time, metaheuristics are tuned by hand in a trial-and-error procedure but different applications like F-Race and paramILS have been proposed in order to customize such procedure. F-Race is a method that is inspired from racing algorithms in machine learning for the offline configuration of parameterized algorithms based on the non-parametric Friedman's two-way analysis of variance by ranks (i.e., Friedman test). F-Race is based on a statistical approach that evaluates a given set of candidate configurations iteratively on a set of problem instances. The statistical evidence is then used to eliminate some candidate configurations and the race continues only with the surviving ones [40].

ParamILS is another approach for parameter tuning on a given class of problem instances that was firs used to automatically configure the CPLEX mixed integer programming solver [41], however it has also been applied to SAT solvers, timetabling, AI planning, etc. ParamILS is an iterative local search algorithm that basically executes two steps until no improvement is found or until some time limit termination criterion is met. First, it tries to find a good parameter configuration in order to evaluate it or it can also start from the default algorithm parameter settings, then it uses local search to explore the parameter configuration space to change some of its parameters and the best known configuration found so far is updated according to some performance criteria [41].

Some other approaches that implement the usage of machine learning for solving combinatorial problems include: MULTI-TAC, a learning system that configures a constraint solver for a particular instance distribution [42] and the Adaptive Constraint Engine (ACE) that learns search heuristics from training instances [43].

2.3.1. Supervised Learning

The previous sections introduced some approaches related to machine learning and combinatorial problems in the context of algorithm portfolios, per-class learning and adaptive algorithms where different machine learning approaches have been used in order to improve algorithms performance for CSPs, SAT and QBF.

The field of machine learning studies algorithms that depend on inputs and feedback from the environment and it's possible to distinguish between three

forms of machine learning: supervised learning, unsupervised learning, and reinforcement learning [44]. However, in this chapter we limit our study to supervised learning, particularly to *classification* using *decision trees* in order to predict a label, based on an array of features.

In supervised learning a system is trained by a *supervisor* given labeled examples where the *learner* is supposed to reproduce such set of examples. Each example is an array of input parameters x called *features* with an associated output label y [45]. The task of supervised learning is to use previous examples in order to build an association $y = f(x)$ between input x and output y respectively.

The output y can be either a label in the case of classification or numerical in the case of regression. *Classification* focuses on the recognition of the class of a specific object described by features x. The output is basically a class $y_i \in \{1, ..., N\}$, while in *regression*, the output is a real number and the objective is to model the relationship between a dependent variable y and one or more independent variables of the array of features x [30].

Both regression and classification are supervised learning problems where there is an input x, an output y, and the task is to learn the mapping from the input to the output ($y = g(x|\theta)$) where g is the model, θ are its parameters and y is a number or a class [30]. Support vector machines, decision trees, and neural networks are among the most common algorithms for classification, however, in this chapter we will limit to describe decision trees which is the most relevant type of classification algorithm for this research:

Decision Trees

Decision trees consist of attributes or features which are used to describe individual cases where each case belongs to exactly one of a set of class labels. Such attributes or features might be continuous or discrete and a decision tree may be either a leaf identified by a class name or a structure of the form $C_i : D_i$, where C_i are logical conditions (e.g., $A < T$, $A > T$, $A = V$ or A in $\{V_i\}$) and D_i is a decision tree or a class name [44]. Below, we show a classical example of a decision tree, using the numerical weather data set of Weka and trained with the J48 algorithm:

```
outlook = sunny
|   humidity <= 75: yes
|   humidity > 75: no
outlook = overcast: yes
outlook = rainy
|   windy = TRUE: no
|   windy = FALSE: yes
```

The decision tree shown before describes the weather forecast for a particular day and the decision (yes/no) to play golf given some particular weather

conditions. The instances of the weather data set have 5 attributes (outlook, temperature, humidity, windy and play). However, the temperature attribute is not part of the decision tree after training with the J48 algorithm.

2.4. Mixed Integer Programming

Integer programming is usually related to the efficient allocation of limited resources to meet a desired objective function with the constraint that the resources can only be divided into discrete parts [46]. Integer programming models are used in a wide variety of applications, including resource assignment, scheduling, planning, auction design, etc. The foundation of integer programming is *linear programming* where a program consists of *variables*, *constraints* and an *objective function*. The variables are unknown quantities or decisions that take numerical values, the constraints must be linear equations over the variables and the objective defines an optimal maximization or minimization assignment to the variables over a linear function [47]. Linear programming models can be continuous, in the sense that decision variables are allowed to be fractional. However, fractional solutions might not be realistic, therefore, *integer programming* adds additional constraints to linear programming by constraining all the variable domains to integer values.

In the context of linear and integer programming, a program is said to be a *mixed integer program or MIP* when some, but not all, variables are restricted to be integer. As in linear programming, the objective function in MIP should be linear on the decision variables and the constraints require that a linear function of the decision variables is either $=, \leq$ or $\geq$ a scalar value. A condition also implies that the decision variables must be non-negative [48].

To model a MIP program, usually a three-step process is required. First, it's necessary to define the decision variables, then the constraints of the model have to be stated in form of linear equations and finally a linear objective function must be defined. An optimal solution in MIP is a feasible solution with the best objective function value. However, sometimes it's usually required to find solutions that just satisfy all the constraints. Given that MIP and CP problems are non-convex, they are frequently solved by a systematic method called Branch and Bound or by alternative methods such as genetic algorithms which are not able to prove optimality. Suppose that we denote $x_1, ..., x_n$ to be a set of decision variables. A linear program would take the form:

$$
\begin{aligned}
\text{Minimize or maximize: } & c_1x_1 + c_2x_2 + ... + c_nx_n \\
\text{Subject to: } & a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n(\leq, =, or \geq)b_1 \\
& a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n(\leq, =, or \geq)b_2 \\
& ... \\
& a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n(\leq, =, or \geq)b_m \\
& x_j \geq 0 \ \ \forall j = 1, ..., n
\end{aligned}
$$

Values $c_j, \forall j = 1, ..., n$ are referred to as objective coefficients, values $b_1, ..., b_m$ at the right hand side of the constraints usually represent available resources or requirements. The values a_{ij} denote how much resource/requirement i is consumed/satisfied by decision j. As explained previously, if the linear program depicted before constraints all the decision variables to be integers, then the problem is said to be an integer linear program; on the other hand, if some (not necessarily all) variables are constrained to be integers, then the problem is said to be a mixed-integer program [48].

2.5. Conclusions

Constraint programming is a paradigm that has developed techniques to address combinatorial problems that usually require parameter tuning and considerable experimentation to offer reasonable solutions. However, the field of machine learning offers a promising approach to improve CP performance by implementing different strategies such as portfolio algorithms and per-class learning that can take advantage of practical applications that usually arise instances that enable them to be solved more efficiently.

The main objective of this research is to improve SABIO's capacity to answer a wider range of instances and machine can be used to improve its performance by learning additional constraints or heuristics. In the following chapter, we manage to introduce the kind of soccer problems that can be addressed with SABIO and in chapter 5 we propose a CP computational model that implements machine learning in order to address such problems.

A constraint programming model has no limitation on the arithmetic constraints while mathematical programming must have specific formulations that satisfy certain properties, e.g., linear equations and inequalities. However, mathematical programming have been studied widely, particularly with outstanding results in planning and scheduling problems, therefore, we also study a mathematical model of SABIO using mixed integer programming model in chapter 6 in order to use a mathematical engine find solutions.

References

[2] M. Wallace, «Practical applications of constraint programming», *Constraints*, vol. 1, no. 1-2, pp. 139–168, 1996.

[3] K. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[5] A. Arbelaez, «Learning during search», PhD thesis, Université Paris, 2011.

[10] J. E. Borrett, E. P. Tsang, and N. R. Walsh, «Adaptive constraint satisfaction: The quickest first principle», in *Proceedings of the 12th ECAI*, Citeseer, 1996.

[11] S. A. M. Elsayed and L. Michel, «Synthesis of search algorithms from high-level cp models», in *Principles and Practice of Constraint Programming–CP 2011*, Springer, 2011, pp. 256–270.

[12] M. Milano and M. Wallace, «Integrating operations research in constraint programming», *Annals of Operations Research*, vol. 175, no. 1, pp. 37–76, 2010.

[13] P. Van Roy and S. Haridi, *Concepts, techniques and models of computer programming*. 2003.

[14] D. Mehta, B. O'Sullivan, L. Kotthoff, and Y. Malitsky, «Lazy branching for constraint satisfaction», in *ICTAI*, Nov. 2013.

[15] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen, «A survey of very large-scale neighborhood search techniques», *Discrete Applied Mathematics*, vol. 123, no. 1, pp. 75–102, 2002.

[16] W. D. Harvey and M. L. Ginsberg, «Limited discrepancy search», *IJCAI (1)*, pp. 607–615, 1995.

[17] M. Dorigo, V. Maniezzo, and A. Colorni, «Ant system: Optimization by a colony of cooperating agents», *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, 1996.

[18] F. Glover, «Tabu search - part i», *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[19] ——, «Tabu search - part ii», *ORSA Journal on computing*, vol. 2, no. 1, pp. 4–32, 1990.

[20] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *et al.*, «Optimization by simulated annealing», *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[21] V. Černỳ, «Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm», *Journal of optimization theory and applications*, vol. 45, no. 1, pp. 41–51, 1985.

[22] P. Van Roy, D. Duchier, L. Kornstaedt, M. Homik, C. Schulte, and T. Müller, *Finite domain constraints*. [Online]. Available: `https://mozart.github.io/mozart-v1/doc-1.4.0/system/node26.html`.

[23] R. Dechter and I. Meiri, *Experimental evaluation of preprocessing techniques in constraint satisfaction problems*. UCLA, Computer Science Department, 1989, pp. 271–277.

[24] R. M. Haralick and G. L. Elliott, «Increasing tree search efficiency for constraint satisfaction problems», *Artificial intelligence*, vol. 14, no. 3, pp. 263–313, 1980.

[25] C. Bessiere and J.-C. Régin, «Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems», in *Principles and Practice of Constraint Programming—CP96*, Springer, 1996, pp. 61–75.

[26] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, «Boosting systematic search by weighting constraints», in *ECAI*, vol. 16, 2004, p. 146.

[27] I. P. Gent, C. Jefferson, L. Kotthoff, I. Miguel, N. C. Moore, P. Nightingale, and K. E. Petrie, «Learning when to use lazy learning in constraint solving.», in *ECAI*, 2010, pp. 873–878.

[28] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997, ISBN: 0070428077, 9780070428072.

[29] A. Smola and S. Vishwanathan, *Introduction to machine learning*. 2008.

[30] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[31] B. O'Sullivan, L. De Raedt, S. Nijssen, and P. Van Hentenryck, Report from Dagstuhl Seminar 11201: Constraint Programming meets Machine Learning and Data Mining, 2011.

[32] L. Kotthoff, «Algorithm selection for combinatorial search problems: A survey», *AI Magazine*, 2014. [Online]. Available: `http://4c.ucc.ie/~larsko/`.

[33] B. A. Huberman, R. M. Lukose, and T. Hogg, «An economics approach to hard computational problems», *Science*, vol. 275, no. 5296, pp. 51–54, 1997.

[34] J. R. Rice, «The algorithm selection problem», 1975.

[35] C. P. Gomes and B. Selman, «Algorithm portfolio design: Theory vs. practice», in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, Morgan Kaufmann Publishers Inc., 1997, pp. 190–197.

[36] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, «Using case-based reasoning in an algorithm portfolio for constraint solving», in *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[37] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, «Satzilla: Portfolio-based algorithm selection for sat.», *J. Artif. Intell. Res.(JAIR)*, vol. 32, pp. 565–606, 2008.

[38] L. Pulina and A. Tacchella, «A multi-engine solver for quantified boolean formulas», in *Principles and Practice of Constraint Programming–CP 2007*, Springer, 2007, pp. 574–589.

[39] M. Birattari and J. Kacprzyk, *Tuning metaheuristics: A machine learning perspective*. Springer, 2009, vol. 197.

[40] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, «F-race and iterated f-race: An overview», in *Experimental methods for the analysis of optimization algorithms*, Springer, 2010, pp. 311–336.

[41] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, «Paramils: An automatic algorithm configuration framework», *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[42] S. Minton, «Automatically configuring constraint satisfaction programs: A case study», *Constraints*, vol. 1, no. 1-2, pp. 7–43, 1996.

[43] S. L. Epstein, E. C. Freuder, R. Wallace, A. Morozov, and B. Samuels, «The adaptive constraint engine», in *Principles and Practice of Constraint Programming-CP 2002*, Springer, 2002, pp. 525–540.

[44] J. R. Quinlan, «Simplifying decision trees», *International journal of man-machine studies*, vol. 27, no. 3, pp. 221–234, 1987.

[45] R. Battiti and M. Brunato, «The lion way», *Machine Learning Plus Intelligent Optimization. Applied Simulation and Modelling*, 2013.

[46] P. R. Watkins, «Integer and combinatorial optimization», *Journal of the Operational Research Society*, vol. 41, no. 2, pp. 177–178, 1990.

[47] R. Bosch and M. Trick, «Integer programming», in *Search methodologies*, Springer, 2014, pp. 67–92.

[48] J. C. Smith and Z. C. Taskin, «A tutorial guide to mixed-integer programming models and solution techniques», *Optimization in Medicine and Biology*, pp. 521–548, 2008.

Chapter 3

Soccer Computational Problems

Contents

3.1. Introduction

Americans usually refer to *Association football* as "soccer" to describe what most people in the world call "football". In fact there is an interesting document by professor Stefan Szymanski [49] where they track the origins of both words and discuss that it should be called football not soccer. Nevertheless, it's not the purpose of this document to argue whether it should be called soccer or football. This section starts by clarifying that it was decided to use the term "*soccer*", in order to refer to the game organized by FIFA whilst the references might use the term "*football*" not only to refer to soccer, but also to American Football as the game organized under the NFL. It's also important to point out that "soccer" is the term adopted for the acronym "SABIO" (Soccer Analysis Based on Inference Outputs, see Chapter 4 for further information).

Soccer competitions offer an excellent opportunity to model interesting problems. These problems are related to questions that soccer fans usually ask about their favourite teams, depending on the tournament or league where their teams participate. A series of articles using combinatorial optimization techniques have been published in order to provide more precise information on the situation of each team within a tournament, for instance, it is possible to determine when a team is mathematically eliminated or when it is the the champion of the competition, also when it is qualified for the next phase, whether it depends only on its own results or on other team results to qualify, and so on.

The purpose of this chapter is to set up a context of common computational problems that can be addressed with SABIO, therefore this chapter is structured as follows. Sections 3.2 and 3.3 briefly review the most relevant problems that can hold in both, a soccer league and a tournament. In section 3.4, an overview of the literature related to the complexity of the soccer elimination problem is presented. Finally, section 3.5 reviews a collection of soccer problems studied in the literature that are relevant to this research.

3.2. League Computational Problems

A league competition consists of n teams playing against each other following a single or double round-robin schedule. For single round-robin, each team i plays team j exactly once and in the double round-robin format, each team i gets to play against team j exactly twice, once as a local team in their home and one more time away.

The games or matches are usually organized in such a way that every team gets to play f number of *fixtures* against the rest of the teams. A fixture involves one or more games between two teams (i and j). Every team is awarded some points depending on the result of the game (3 points – win, 1 point – tie, 0 points – loss). Teams are ranked by total points (i.e., the addition of the initial points and the points obtained in all the matches), and at the end of the league, the team with the highest amount of points is declared as the champion.

At some stage of the league competition when there are still some remaining games play, one may ask if a particular team is eliminated or has a theoretical chance to become the champion. This question refers to a problem that is also known as the *elimination problem* and in the case of a league, this question basically asks if team i can no longer become champion, which can only be achieved by reaching first position. This problem and its complexity will be covered in section 3.4.

If a team is already eliminated, some fans might also be interested to know if their teams have a chance to finish in another particular position like second or third place, since these positions can bring some benefits, for instance, the top teams (first, second and third) at the BBVA league qualify to the UEFA Cham-

pions League, so other interesting questions are "Can team i finish in position k? can team i finish in a better position than k?" these questions are more general than the elimination problem since you can check whether a team is eliminated or not by asking if team i can finish in position $k = 1$. Problems related to this questions will be covered in section 3.5.

3.3. Tournament Computational Problems

A tournament competition consists of n teams and as the league, the pointing rules are the same (3 points – win, 1 point – tie, 0 points – loss). Some soccer tournaments are usually organized in two stages:

- **The qualification stage:** this stage can be organized with a round-robin schedule where each team i plays against every other team j an equal number of times (usually one or two). The games can also be organized in f number of *fixtures* where teams play against others in one or more matches. When the qualification stage is finished, every team is ranked by using the points they sum up during the tournament and then the first k teams with the highest scores, qualify to the playoffs which is the next stage.
- **The playoffs stage:** The playoffs might be played in different forms, usually as a single elimination or "knockout" or as a small tournament.

The qualification stage raises important questions such as "Will my team qualify to the playoffs? how many points does my team need in order to qualify?". Problems related to this kind of questions will be covered in section 3.5.

3.4. The Elimination Problem Complexity

At a particular stage of a baseball/soccer/basketball/handball season, the elimination problem consists of determining if a team i is already eliminated and cannot become champion or tied for first place for all the possible outcomes of the remaining games, given that team i has w_i wins and g_{ij} games left to play against team j. The complexity of this problem depends on the way how scores are allocated according to the outcome of every match [9]. This elimination problem could be transform into a network flow problem and would be polynomially solvable, if the old FIFA rule (0, 1, 2) was used, nevertheless, Kern and Paulusma [9] [50] and Bernholt et al. [8] independently proved that for the actual pointing rule (0, 1, 3) the problem is NP-complete.

In this section two scenarios will be considered. In the first scenario (section 3.4.1), the elimination problem can be solved in polynomial time like in baseball and in the old FIFA scoring system for soccer. In the second and most relevant

scenario for this document (section 3.4.2), the problem becomes NP-complete under the actual FIFA scoring system for soccer:

3.4.1. The Elimination Problem in Baseball and the Old FIFA Scoring System for Soccer

The elimination problem is a classical computational problem that has already been studied since the 1960's and first popularized by Alan Hoffman, particularly in baseball where there are no ties and the games are played under the *"1-point-rule"*, i.e., the winner gets 1 point and no ties are allowed. There have been different studies about the elimination problem in baseball, for instance Schwartz [51] proposed a method using maximum network flows in order to determine if a single team is eliminated. *These early solutions to the elimination problem in baseball using network flows resulted in polynomial time algorithms.*

Hoffman and Rivlin [52] generalized Schwartz proposal and gave a necessary and sufficient condition for not being eliminated from finishing in k-th place. Later on, McCormick [53] showed that *determining whether a team is eliminated from finishing the season in k-th place or better is NP-Complete for baseball.* Adler et al. proposed in [54] an integer programming formulation to compute all the eliminated teams of a baseball tournament at certain stage. Wayne [55] proposed a structural property for the baseball elimination and ordered the teams according to their total number of possible wins and showed that if a team is eliminated, then so are all teams below it in the ordering. He showed that there is a threshold value W^* for all teams i with initial points w_i and remaining games g_i, and it holds that team i is eliminated if and only if $w_i + g_i < W^*$, i.e. i can not win W or more games. This approach brought faster algorithms since it allows to compute all eliminated teams simultaneously.

Bernholt et al. [8] pointed out that the old pointing system in soccer (two points to a win, one point to a tie and zero points to a loss), could be treated as a special instance of the baseball elimination problem where one game between teams i and j could be seen as two games between teams i and j under the 1-point-rule of baseball (without ties) and then the problem could be solved in polynomial time, using network flows as proposed by Wayne [55].

3.4.2. The Elimination Problem in Soccer and the New FIFA Scoring System

In soccer and before 1995, the traditional pointing rule was to assign two points to a win, one point to a tie and zero points to a loss. After 1995, FIFA formally decided to increase from two to three the number of points assigned to the winning team and became standard in international tournaments, as well as most national soccer leagues, hoping that with an increase in the reward for victory, teams would become more offensive [56].

In 1999, Bernholt et al. [8] specified that for every game between i and j in a soccer/basketball/handball season, consisting of a set $\{1, 2, ..., N\}$ of teams, points are awarded according to some "(α, β)-rule". This notation means that the winner team gets α points and the losing team gets 0 points. The β points are awarded for both teams if the game ends up in a tie.

Now a days, soccer is played under the (3,1)-rule or the *"3-point-rule"* and Bernholt et al. [8] showed that soccer elimination problem is hard to decide under this pointing system and proved that *the soccer elimination problem is NP-Complete if all teams have at most three remaining games.* In the same article, they also proved that the problem can be solved in polynomial time if each team has at most two remaining games.

It is clear that soccer elimination under a (3,1)-rule is NP, since it's possible to guess the outcomes of the remaining games and compute the final ranking [8]. To prove NP-Completeness, Bernholt et al. [8] managed to represent an instance of EFEP (European Football Elimination Problem) as an undirected multigraph and then showed that for each input formula to 3-SAT, it is possible to construct a labeled multigraph that is satisfiable only if the team for which you want to decide whether is eliminated or not, can still become champion.

As in baseball, Gusfield and Martel showed that there is a threshold value W^* for all teams i with initial points w_i and remaining games g_i, and it holds that team i is eliminated if and only if $w_i + g_i < W^*$, nevertheless, calculating such threshold for soccer might be NP-hard [57][8].

A more generalized study of complexity aspects of sports competitions (including soccer) was proposed by Kern and Paulusma [9][50], who used the triple (α, β, γ) to denote the points assigned to a team when participating in a match. The score is increase by $\alpha \in R$ if the team loses, by $\beta \in R$ if the game ends in a tie and by $\gamma \in R$ if the team wins and they always assumed that $\alpha \leq \beta \leq \gamma$. They determined the computational complexity of the sports elimination problem assuming that $(P \neq NP)$ and proved that a game played under the (α, β, γ) rule is polynomially solvable for three cases:

(I) $\alpha = \beta$

(II) $\beta = \gamma$

(III) $\alpha + \gamma = 2\beta$

in all other cases, like in the case of soccer competitions with score allocations $(\alpha = 0, \beta = 1, \gamma = 3)$ Kern and Paulusma proved NP-Completeness by a reduction from three-dimensional matching to a particular multigraph model $G = (V; E)$ that from a very general view, the vertices correspond to teams and edges are in 1–1 correspondence with remaining matches [9][50].

3.5. Other Problems related to Soccer

Promotion and Relegation: Soccer fans are also interested to check whether their teams can be either promoted or relegated at certain stage of a tournament, that is, the top l teams in ranking from a division of a league or tournament, are promoted to a higher division, while the last l teams in ranking from a higher division are relegated to a lower one. Kern and Paulusma [9] also showed that questions like "is there a chance that team i ends up with the lowest final score?" or "is there a chance that team i ends up being one of the three teams that have the three lowest final scores?", which might cause a direct relegation to a lower division for a particular team i, turn out to be NP-Complete and they used a version of the elimination problem to prove NP-Completeness for the rule $(\alpha = 0, \beta = 1, \gamma = 3)$ applied in soccer.

The Elimination Number: The elimination number problem relates to the minimum number of points that a team must get in order to have a chance of finishing in first place or to secure a place in the playoffs. In other words, it can be simplified as the following question: if team i is not eliminated, what is the minimum number of points that i can win and still at least tie for first place or qualify to the playoffs? Gusfield and Martel showed that computing such number of points for soccer is NP-Hard and that it can be as hard as a subset sum problem [57].

Guaranteed Point Placement: Given a soccer tournament under the 3-point-rule system, the guaranteed point placement determines if it is possible that any assignment of outcomes to the remaining games will place team t at position at least K in the final standing [58]. Christensen et al. [58] showed that the decision problem of whether or not a team is guaranteed a certain minimum position is coNP-Complete by using a reduction from 3-SAT to a graph instance of the Guaranteed Point Placement problem.

Guaranteed Qualification Problem: This problem consists of calculating the minimum number of points any team has to win in order to be qualified, given any results of the rest of the teams. This problem depends on the current number of points of every team in the standing table and on the missing games to be played. This problem was tackled by Riberio and Urrutia [59] using integer programming. They calculated a guaranteed qualification score (GQS) and their implementation can also be used to determine if a team is mathematically qualified to the playoffs, if and only if its number of points is greater than or equal to its GQS.

Possible Qualification Problem: This problem consists of calculating the amount of points each team has to win in order to have any chance to be qualified.

This problem was studied by Riberio and Urrutia [59] using integer programming for which they calculated a possible qualification score (PQS) which also depends on the current number of points of every team in the standing table and on the missing games to be played. Analogously, to their Guaranteed Qualification Problem, this model can be used to check if a team is mathematically eliminated from the playoffs when the total number of possible points(i.e., the number of remaining games multiplied by three) plus the number of points that the team already has is less than its PQS.

The Score Vector Problem: Kern and Paulusma [50] suspected that deciding if a score vector is reachable or not is a difficult problem. In 2009, Pálvölgyi denoted the problem of deciding whether a given score vector is a possible result of a soccer-tournament or not as the Score Vector Problem and proved that the problem is NP-complete [60] by using a reduction from 3-SAT for the cases like the actual pointing rule ($\alpha = 0, \beta = 1, \gamma = 3$) applied in soccer.

3.6. Conclusions

In this chapter we over viewed some common computational problems that can be addressed with SABIO and showed how soccer computational problems complexity is highly related to the rule system used to determine the points in a game. We also showed that most of the common soccer questions that people wonder about their teams, turn to be NP-Complete for the actual three point rule system (3 points – win, 1 point – tie, 0 points – loss).

Identifying teams that are already eliminated form first place is a problem that has been widely used to show the application of linear programming and network flows [57], in fact, there have been many linear programming models specially for baseball. In 2002, a baseball Web site was created as a component of the Berkeley RIOT project (Berkeley Remote Interactive Optimization Testbed), in order to broadcast optimization-based information of the Major League Baseball (MLB) [54]. To create the web page, they first generated a set of mathematical models using CPLEX for calculating the new play-off statistics, and then they developed a software system to produce automated updates of the Web site. Similarly, back in 2005, the FutMax system and its framework described in [59][61] provided detailed information of the conditions of qualification and elimination of of Brazilian national soccer and some other competitions based on mathematical models implemented in CPLEX.

References

[8] T. Bernholt, A. Gälich, T. Hofmeister, and N. Schmitt, «Football elimination is hard to decide under the 3-point-rule», in *MFCS*, 1999, pp. 410–418.

[9] W. Kern and D. Paulusma, «The new fifa rules are hard: Complexity aspects of sports competitions», *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 317–323, 2001.

[49] S. Szymanski, *It's football not soccer*, URL: `http://ns.umich.edu/Releases/2014/June14/Its-football-not-soccer.pdf`, University of Michigan, 2014. [Online]. Available: `http://ns.umich.edu/Releases/2014/June14/Its-football-not-soccer.pdf`.

[50] W. Kern and D. Paulusma, «The computational complexity of the elimination problem in generalized sports competitions», *Discrete Optimization*, vol. 1, no. 2, pp. 205–214, 2004.

[51] B. L. Schwartz, «Possible winners in partially completed tournaments», *SIAM Review*, vol. 8, no. 3, pp. 302–308, 1966.

[52] A. Hoffman and T Rivlin, «When is a team "mathematically" eliminated?», Princeton, NJ: Princeton Symposium on Mathematical Programming, 1967, pp. 391–401.

[53] S. T. McCormick, «Fast algorithms for parametric scheduling come from extensions to parametric maximum flow», *Operations Research*, vol. 47, no. 5, pp. 744–756, 1999.

[54] I. Adler, A. L. Erera, D. S. Hochbaum, and E. V. Olinick, «Baseball, optimization, and the world wide web», *Interfaces*, vol. 32, no. 2, pp. 12–22, 2002.

[55] K. D. Wayne, «A new property and a faster algorithm for baseball elimination», *SIAM Journal on Discrete Mathematics*, vol. 14, no. 2, pp. 223–229, 2001.

[56] J. C. Guedes and F. S. Machado, «Changing rewards in contests: Has the three-point rule brought more offense to soccer?», *Empirical Economics*, vol. 27, no. 4, pp. 607–630, 2002.

[57] D. Gusfield and C. Martel, «The structure and complexity of sports elimination numbers», *Algorithmica*, vol. 32, no. 1, pp. 73–86, 2002.

[58] J. Christensen, A. N. Knudsen, and K. S. Larsen, «Soccer is harder than football», *International Journal of Foundations of Computer Science*, vol. 26, no. 04, pp. 477–486, 2015.

[59] C. C. Ribeiro and S. Urrutia, «An application of integer programming to playoff elimination in football championships», *International Transactions in Operational Research*, vol. 12, no. 4, pp. 375–386, 2005.

[60] D. Pálvölgyi, «Deciding soccer scores and partial orientations of graphs», *Acta Univ. Sapientiae, Math*, vol. 1, no. 1, pp. 35–42, 2009.

[61] C. Lucena, T. Noronha, S Urrutia, and C. Ribeiro, «A multi-agent framework to retrieve and publish information on qualification and elimination data in sports tournaments», *Rio de Janeiro, Brazil: Monografias em Ciência da Computação*, 2006.

Chapter 4

SABIO as a CP Application for Soccer Analysis

Contents

4.1. Introduction

Soccer is one of the most popular sports in the world with a big impact in mass media and followed by millions of people. Frequently, when a tournament or a league is getting to an end, sports media offer statistics and make untested statements about the qualification conditions of the teams. However, as shown

in Chapter 3, these kind of assertions might not be as trivial as they think because the combinatorial options are usually big and require expertise and certainly some amount of computation in order to make true statements about the teams conditions.

One of the goals of this research (i.e., *G2* depicted in section 1.2) relates to identify a family of instances where SABIO presents low performance in order to train a machine learning model to improve its efficiency. This chapter presents in section 4.2 a complete introduction to SABIO as a CP application including the computational problems that can be addressed with the application and the type of queries that users can define. In the same section, we also include a brief overview of the variable/value heuristics implemented in SABIO to explore the search space. In section 4.3 we managed to identify hard instances for the former version of SABIO for which we run 3 sets of empirical tests and later analyse the obtained results in section 4.4.

4.2. SABIO as a CP Application

SABIO stands for *Soccer Analysis Based on Inference Outputs*) and it is a CP platform created with the academic support of the AVISPA research group from the Universidad del Valle (`http://avispa.univalle.edu.co/`).[1] SABIO can be used to discover information and answer questions related to soccer, for instance:

- Is my team already qualified?
- Will team i qualify with 30 points?
- Can my team be within the first 8 positions if it loses against team x?

As mentioned before, soccer competitions bring an excellent opportunity to model interesting problems. Previous approaches like Futmax were based on a 3 phase framework: (1) collection of game results from one or more sources, (2) solution of one or more combinatorial optimization problems, and (3) publication of information on each team via the Internet [61]. SABIO differs from this approach because it offers an interactive platform that soccer fans can use to model different scenarios in order to ask *user defined questions* about their teams, independently of the tournament or league, except for the default pointing system (i.e., the three point rule).

4.2.1. Addressing Computational Problems With SABIO

SABIO checks for possibilities related to teams in any soccer tournament based on points and it was created in order to be used by any person (No CP or tech-

[1] SABIO translates "a wise person" in Spanish. The former online application can be accessed at `www.sabiofutbol.com`.

nical knowledge is required). SABIO helps its users discover information related to soccer teams from a particular tournament and offers graphical interfaces to impose different kind of assumptions in form of constraints (See figure 4.1). These constraints allow users to create different queries to represent soccer scenarios and we have categorized them in four types of constraints: position in ranking (i.e., Type 1 constraints for short), relative position (i.e., Type 2), game results (i.e., Type 3) and final score (i.e.,Type 4):

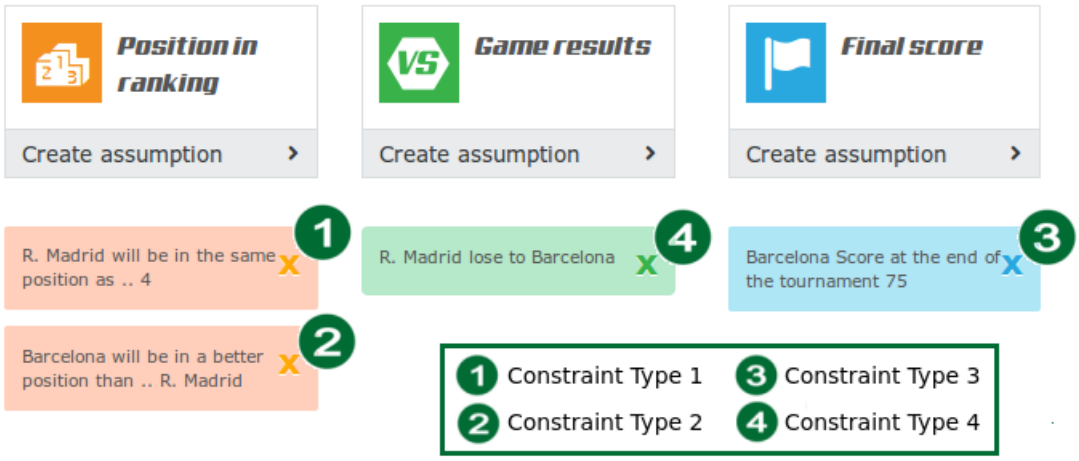

Figure 4.1: SABIO GUI to impose constraints type 1, 2, 3 and 4.

Type 1 Constraints - Position in Ranking. These kind of assumptions can be used to impose constraints about the positions of the teams at the end of a tournament. Table 4.1 depicts all the options that constrain a team to a position by using a logical operator ($<, >, \leq, \geq, =$):

Table 4.1: Position in Ranking Constraints

Team	Constraint	Position
R.Madrid	Will be in a better position than...	3
R.Madrid	Will be in a better or same position as...	4
R.Madrid	Will be in the same position as...	1
R.Madrid	Will be in a worse position than...	2
R.Madrid	Will be in a worse or same position as...	5

Type 2 Constraints - Relative Position. These kind of assumptions can be used to impose constraints about the positions of the teams respect to others. Table 4.2 outlines all the options that constrain a team position to another by using a logical operator ($<, >, \leq, \geq, =$):

Table 4.2: Relative Position Constraints

Team	Constraint	Team
R.Madrid	Will be in a better position than...	Barcelona
R.Madrid	Will be in a better or same position as...	Barcelona
R.Madrid	Will be in the same position as...	Barcelona
R.Madrid	Will be in a worse position than...	Barcelona
R.Madrid	Will be in a worse or same position as...	Barcelona

Type 3 Constraints - Final Score. These kind of assumptions can be used to impose constraints about the expected final score of the teams at the end of a tournament. Table 4.3 depicts the option that correspond to a team constrained to a final score:

Table 4.3: Final Score Constraints

Team	Constraint	Score
Barcelona	Score at the end of the tournament	75
R.Madrid	Score at the end of the tournament	70

Type 4 Constraints - Game Results. These kind of assumptions can be used to impose constraints about matches results in any fixture. Table 4.4 shows the three possible outcomes that a math can be constrained to:

Table 4.4: Game Result Constraints

Team	Constraint	Team
Barcelona	beats	R.Madrid
Barcelona	loses to	R.Madrid
Barcelona	ends in a tie with	R.Madrid

These four kind of constraints can be used to tackle different soccer problems like the ones outlined in Chapter 3, for instance:

- Type 1 constraints can be used to create scenarios to tackle the **elimination problem** and ask if a particular team is eliminated from a league or has a theoretical chance to become champion (e.g., R.Madrid will be in position 1). If SABIO finds a feasible solution where R.Madrid finishes first in the standing table, it means that the team still has a chance to become the champion, otherwise, the team is already eliminated.

- Type 1 constraints can also be used to check if a team has chances to **qualify to the playoffs** (e.g., R.Madrid will be in a better or same position as 8), similarly for queries related to **promotion and relegation**.
- It's also possible to address the **score vector problem** by using Type 3 constraints to define the scores of every team, in order to check if it is a possible outcome of the competition.
- Iterating over a combination of Type 1 and Type 3 constraints can also be useful to tackle the **guaranteed qualification problem** or calculating the minimum number of points in order to guarantee qualification. (e.g., R.Madrid will be in a worst position as $8 \wedge$ R.Madrid score at the end of the tournament is 70). If SABIO finds a feasible solution where R.Madrid finishes worst than 8^{th}, it means that 70 points are not enough to qualify in some scenarios, then the final score has to be increased and run the query again until finding a scenario with an unfeasible answer. Despite the fact this is not an efficient way to address this problem, the MIP version of SABIO proposed in Chapter 6 will make it more efficient.
- Type 2 and 4 constraints, might be of interest to the users to add suppositions about the teams placement or possible outcomes of the matches in any of the problems mentioned above (e.g., R.Madrid will be in position 1, winning the local games and losing the rest of the matches.)

The former version of SABIO allows to address the problems mentioned above, however, in section 4.3 we managed to identify a family of hard instances. The following numerals offer a general description of the former CSP of SABIO including an overview of its variables and variable/value selection heuristics. Refer to section 5.2 for a detailed description of the model including a series of improvements using machine learning.

4.2.2. Overview of the Former Version of SABIO

When SABIO gets a query like the ones mentioned in the previous numeral, it creates a set of variables for every team in order to model the matches to play [62]. Here we present a general description of such variables:

- **Points per match:** these variables help to model every match between two teams. Each team has associated a set of variables (i.e., one variable per match), therefore, a match is modeled with two variables (i.e., one for each team) where the variable of the winning team gets bounded to 3 and the variable of the losing team gets bounded to 0. In case of a tie, both variables get bounded to 1.

- **Total points:** these variables represent the amount of points that every team will have at the end of the competition. Each team has associated one total points variable and it is basically the addition of the *points per match* associated to the team plus its initial points.
- **Position:** Every team has associated a final position variable that depends directly on the total points of the team.
- **Booleans to determine the position:** Every team has associated an array of boolean variables that is used to compare its total points to other teams, in order to determine its position. The values that every variable of the array can take are: 1 if the team that is compared to has more points or 0 otherwise. The position is calculated with the addition of every variable of the array.

Former Branching Heuristics:

In early versions of SABIO, the search was configured with a *first fail* strategy and the *points per match variables* that represent the games of the teams were selected lexicographically according to the teams names (i.e., all the points variables of the first team in the list, then all the variables of the second team, and so on). If a competition has different teams (e.g., A, B, C, ...), each one with n matches to play, then the top of the tree for variable selection would generally correspond to the one shown in figure 4.2.

Given that the search was driven by a first fail strategy, then it assumed that the teams would first lose the games, then tried with ties and only after that, tried winning the soccer matches because the first fail strategy selects the leftmost variable with the smallest domain size and creates a choice point with the lower bound of the domain $\{0, 1, 3\}$.

In [62] Vargas shows how the value selection strategies (min, mid, max) or (lose, tie, win) respectively, affect the search tree and also the time to find a feasible solution. The author also depicts how the time to find an answer is highly related to the instance that is pretended to be solved, just like in many documented constraint programming problems.

In 2014, a modification of the early branching heuristics of SABIO was proposed in order to improve the general performance, based on a series of tests [62]. The new branching heuristics allowed to define the way how *the variables of the teams and their corresponding values were selected depending on the user defined constraints.* An approximation of the implemented improvements include the following variable/value selection heuristics:

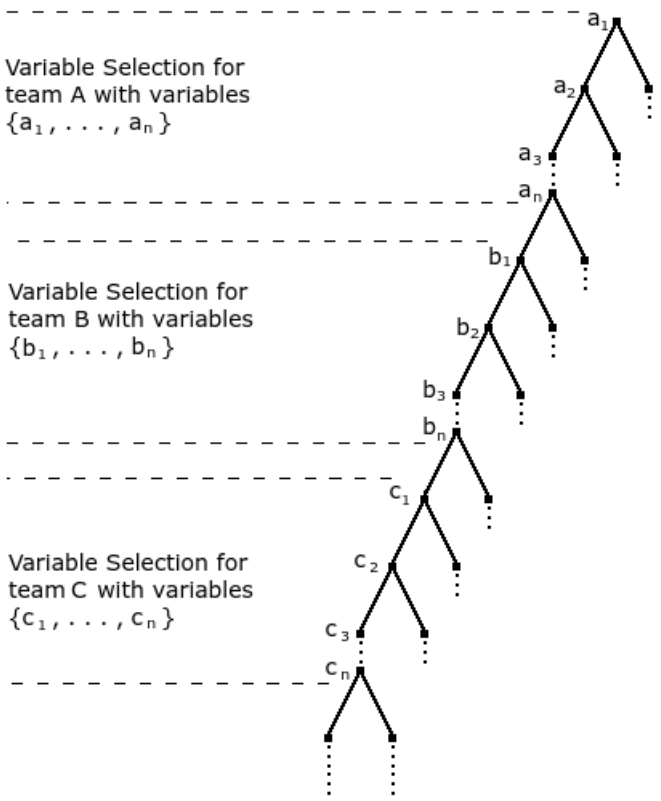

Figure 4.2: Former teams' variable selection for a competition.

Variable selection:

The application was improved with a fixed variable selection that determines how to select the teams' variables offline. This variable selection is based on a system of priorities for the teams involved in constraints Type 1 (Position in ranking) and Type 2 (Relative position). Let pri_i denote an non-negative integer that represents the priority to select the variables of team i during branching, so if team i does not appear in any query, then $pri_i = 0$. Tables 4.5 and 4.6 outline the heuristics for variable selection based on a priority system in a competition with N teams.

Value selection:

The application was improved with a strategy that determine the form to start assigning the points to the teams. The author proposed some heuristics based on the analysis of assumptions that included the operators $\{>, >=, <, =<\}$ for the teams involved in constraints Type 1 (Position in ranking) and Type 2 (Relative position). Let str_i denote the global branching strategy for team i (i.e., $str_i \in \{win, tie, lose\}$), in such way that str_i starts with "tie" as a default value. Tables 4.5 and 4.6 outline the implemented heuristics for value selection.

Table 4.5: Heuristics for Type 1 Constraints (Position in Ranking Constraints)

Team	**Constraint**	**Position**	**Heuristic**
i	Better/ Better or same position as	Position	$str_i = max \wedge pri_i = (N - Position)$
i	Worse/ Worse or same position as	Position	$str_i = min \wedge pri_i = Position$

A position in ranking constraint (Type 1) consists of a team compared to a final position (e.g., R.Madrid will be in a better position than 3). Table 4.5 shows that assumptions that specify that a team has to reach "Better" positions, indicate that the team has to start winning the games ($str_i = max$) in order to satisfy the constraint. On the other hand the priority $pri_i = (N - Position)$, indicates that the closer the final position is to the top teams in the standing table, the higher is the priority to select such team variables. Similarly when assumptions indicate that a team has to reach "Worst" positions, the team has to start losing the games ($str_i = min$) and the priority $pri_i = Position$, indicate that the closer the final position is to the bottom teams in the standing table, the higher is the priority to select such team variables.

Table 4.6: Heuristics for Type 2 Constraints (Relative Position)

Restricción	**Heuristic**
i Better/ Better or same position as j	$MAX = maximum(pri_i, pri_j)$ $pri_i = MAX + 2 \ \wedge \ str_i = max$ $pri_j = MAX + 1 \ \wedge \ str_j = min$
i Worse/ Worse or same position as j	$MAX = maximum(pri_i, pri_j)$ $pri_i = MAX + 2 \ \wedge \ str_i = min$ $pri_j = MAX + 1 \ \wedge \ str_j = max$

A relative position constraint (Type 2) consists of a team final position compared to another team (e.g., R.Madrid will be in a better position than Barcelona). Table 4.6 shows that assumptions specifying that team i finishes in a "Better" position than j, indicate that team i has to start winning the games ($str_i = max$) while j starts loosing them ($str_j = min$). In order to determine the variable branching priorities, it's necessary to find the highest priority between i and j using a function $maximum(pri_i, pri_j)$ and then reassign the priorities to both teams adding 1 and 2 to break ties. Similarly happens with constraints indicating that team i finishes in a "Worse" position than j.

Finally, the proposed heuristics in [62] applies a value selection strategy to the teams that are not included in the constraints. If the amount of constraints indicating "Better" positions is greater that the amount of constraints indicating "Worse" positions then the remaining teams are assigned a *min* strategy, otherwise they are assigned a *max* strategy. Picture 4.3 illustrates the top of the tree for variable selection based on the priority system for user defined constraints depicted on tables 4.5 and 4.6.

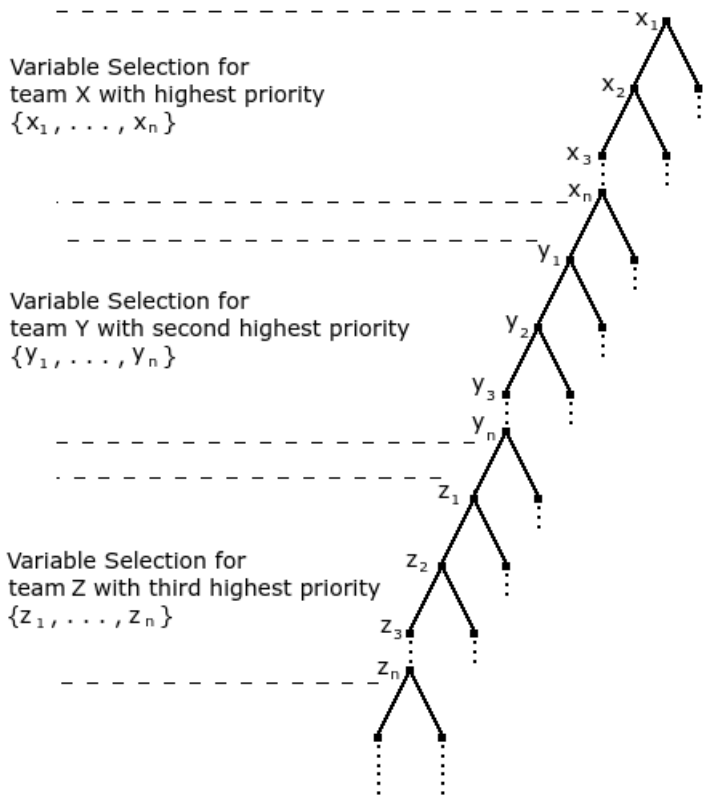

Figure 4.3: Variable selection tree based on priorities for user defined constraints.

4.2.3. Search Space Size

As mentioned before, constraint programming problems involve exploring a search space for solutions. In real world applications these spaces are usually too big to be completely explored by using exhaustive search. Most of these problems are NP-Complete and to offer a reasonable solution, they require considerable experimentation, complex algorithms and data structures[6]. SABIO determines the final positions tables of a soccer competition which is a combinatorial problem that depends on:

- The number of games per fixture [P].

- The number of fixtures to play [F].
- The number of possible results per match, which in soccer are 3:
  1. Team A wins, Team B loses
  2. Team A loses, Team B wins
  3. Team A ties, Team B ties.

According to the previous scenario, the total number of possibilities is given by the expression 3^{P*F} [62] which makes SABIO a constraint problem where exploring all the possible possible combinations is not a good option. If we consider a league with 5 fixtures to be played and 9 games per fixture (i.e., 18 teams in total), then it is equivalent to $2.95 * 10^{21}$ possible scenarios.

SABIO implements a static distribution that would order the variables and define their search strategy before starting the actual search. As mentioned in [7], at the beginning of CP many *static search strategies* were considered, but better results are obtained when *dynamic ordering strategies* are introduced. These strategies compute the next variable to assign during search by using a criteria measurement and the variable having the best value for this criteria is then chosen; the next numeral explores how the search algorithm heuristics for variable/value selection can be configured in SABIO, in order to get better results.

4.2.4. SABIO Search Algorithm Configuration

Apart from the heuristics for variable/value selection depicted in subsection 4.2.2, SABIO also offers a module with advanced options that require some expertise from the users to configure the way how the search can take place when executing a query. This module can be used for those cases in which the predefined heuristics don't find a feasible solution in a particular amount of time. This advanced search options are shown in figure 4.4:

As mention in section 2.3, algorithm configuration can be treated as an optimization problem and in 2007, Hutter et al [63] presented an iterated local search (ILS) algorithm for algorithm configuration to minimise run-time or for maximising solution quality in optimization problems (paramILS). Another approach for parameter tuning using statistical techniques was introduced by using F-Race [40].

This option where the user can customize the way how the search tree is explored, was a first attempt to improve SABIO's performance when the solver doesn't find feasible solutions in the estimated time. However, it requires that the user understands how the search algorithm works in order to obtain good results and potential users of SABIO might not have such kind of understanding.

The module shown in figure 4.4 allows to order the teams in any particular form. This team arrangement is used to determine the variable selection order in which the branching algorithm selects the variables of the teams. The module

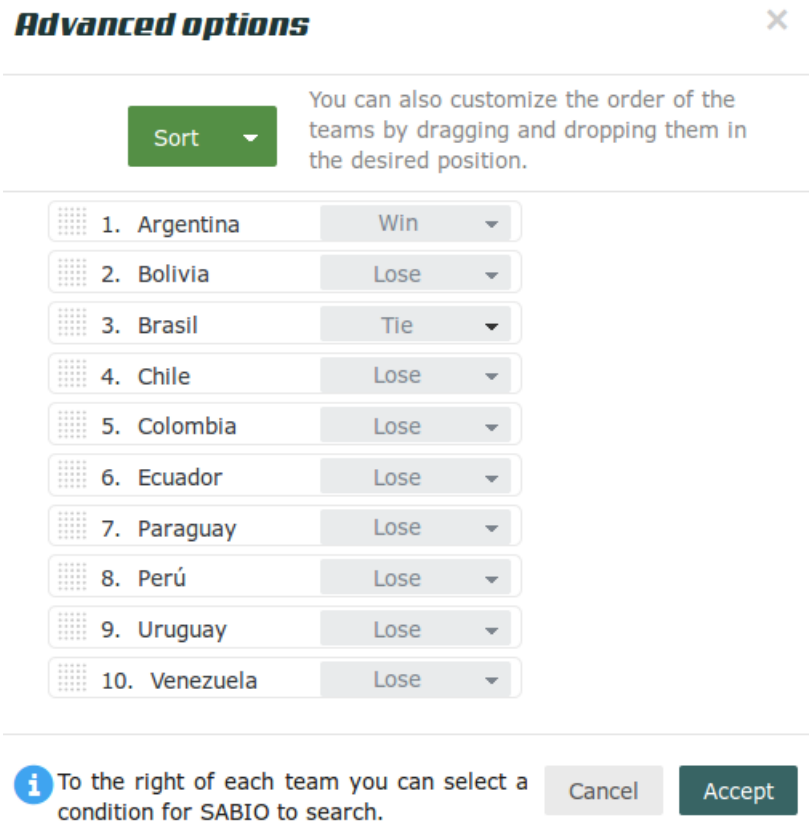

Figure 4.4: SABIO GUI to configure variable/value selection.

also allows to select the default value of the team variables at the beginning of the branching (i.e., win, tie, lose). Given that a soccer competition has N teams, there are $N!$ combinations for the teams' order, also every team has 3 possible value selection options that corresponds to 3^N forms to configure the value selection. Therefore, the module can be configured to $N! * 3^N$ forms to explore the search tree, that is $2.48041 * 10^{24}$ possible combinations for a competition with 18 teams. Given the huge size of the possible forms to customize the search algorithm, the probability of success for a naive user is very low.

4.3. Identifying Hard Instances for SABIO

In order to identify hard instances for the former version of SABIO, depicted in the previous section, we run 3 sets of empirical tests. The first set of tests was applied over 3 different type of constraints, namely, Type 1 or position in ranking, Type 2 or relative position and Type 3 or final score (See section 4.2.1 for further information about the constraints). Type 4 or game result constraints were not tested because they don't require a search process when evaluated independently.

The second set of tests was applied over Type 1 constraints due to the low performance results obtained in the first test. Finally, the third set of tests was applied to analyze the impact of the execution time limit on the solver.

We highlight that the former version of SABIO was implemented using Mozart-Oz (V 1.4.0) as our CP reference solver and every test was executed in a 4-core machine, featuring an Intel Core i5 processor at 2.3 Ghz with 4096 MB of RAM.

4.3.1. First Set of Tests (Evaluation per Constraint Type)

For this particular set of tests we focused our attention in the Colombian league (liga Postobón 2014-I) with 18 teams and 18 fixtures to play in a single round-robin schedule. We provided experiments for five scenarios by exploring different stages of the competition (i.e., fixtures 7, 9, 11, 14, and 16) because the objective was to identify the complexity level of the queries as the tournament unfolds.

For each fixture we created pseudo-random instances with position in ranking queries (Type 1), relative position queries (Type 2), and final point queries (Type 3). Game result constraints (Type 4) were not evaluated because they don't require a search process and can be trivially solved when evaluated independently. For every query type (Type 1, 2 and 3) and fixture (7, 9, 11, 14, 16), a series of instances with different number of suppositions were created (100 with 2 suppositions, 100 with 3 suppositions, and the same for 4, 5, 7, and 9 suppositions) to feature a total of 9000 instances.

Test Configuration

In order to run this test, SABIO was configured to use a time limit of 30 seconds to find feasible answers in each instance. We recall that SABIO is a Web based application and long answer times are not desirable, so we pretended to evaluate the experience that a general user might have by interacting with the application.

Test Results

Table 4.7 shows the number of unsolved instances within 30 seconds for the three type of queries with different number of suppositions at different stages of the competition:

	Type 1						Type 2						Type 3					
Fixture \ Sup.	2	3	4	5	7	9	2	3	4	5	7	9	2	3	4	5	7	9
7	24	28	38	46	50	52	-	-	-	-	-	-	-	2	-	-	6	3
9	23	28	35	46	49	44	-	-	-	-	-	-	-	-	-	-	1	1
11	25	27	31	44	49	47	-	-	-	-	-	-	-	-	-	-	-	-
14	23	33	38	41	51	45	-	-	-	-	-	-	-	-	-	1	-	-
16	23	31	29	31	25	13	-	-	-	-	-	-	-	-	-	-	-	-
Subtotals	118	147	171	208	224	201	-	-	-	-	-	-	-	2	-	1	7	4
Totals	**1069**						**0**						**14**					

Table 4.7: Test 1 - Number of unsolved instances for constraints Type 1, 2, and 3 during 30 seconds.

In general we can observe that Type 1 or position in ranking queries (e.g., R.Madrid will be in a better position than 5) have the highest amout of unsolved instances. Keeping in mind that 3000 instances were created for each query type (5 fixtures * 6 supposition categories * 100 instances), then 1069 unsolved instances correspond to 35.63% of the total number of Type 1 queries. Type 2 queries were solved in its whole while for Type 3 queries, a total of 0.47% were not solved, from which 11 of them happened to be in fixture 7.

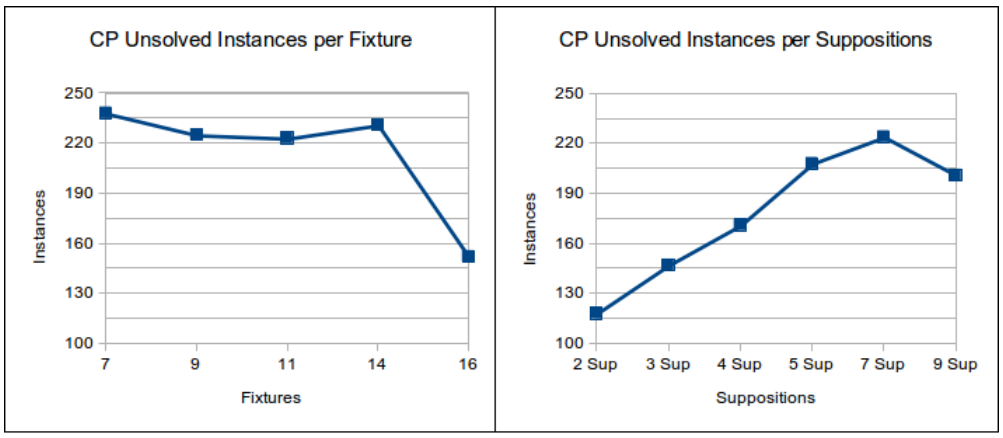

Figure 4.5: Test 1 - Unsolved Type 1 queries per fixture and number of suppositions.

Interestingly, we can observe in figure 4.5 that the amount of unsolved instances per fixture tends to decrease as the league unfolds, this behaviour might attributed to the fact that the search space is bigger at the beginning of the league than it is at the final stages. On the other hand, we can also observe an increment in the amount of unsolved instances as the number of suppositions grow. This last behaviour might be natural since the complexity of the problem grows as we add constraints, however, in the following tests we will analyse this behaviour because

it can also be caused by a particular type of queries that becomes more frequent as the number of suppositions increases.

4.3.2. Second Set of Tests

Due to the results obtained in the previous test, we will focus the following experiments in constraints Type 1 because they represented the highest percentage of unsolved instances. For the previous test, different queries were created for each of the fixtures (i.e., 7, 9, 11, 14, and 16). This test pretends to evaluate the behaviour of SABIO using the same set of instances for all the fixtures, in order to *identify difficult suppositions to solve, depending of the state of the league*. It was also decided to add one more fixture (fixture 5) in order to observe the behaviour of the search at the very beginning of the competition.

Test Configuration

This test was executed using the instances from fixture 16 created in the previous section and used for the first test. The set of instances was evaluated for fixtures 5, 7, 9, 11, and 14 in order to identify the behaviour of SABIO over the same set of queries for which every execution was configure with a time limit of 30 seconds.

Test Results

Table 4.8 registers the number of unsolved instances within 30 seconds for a set of Type 1 queries with different number of suppositions at different stages of the competition:

	Type 1						
Fixture \ **Sup.**	**2**	**3**	**4**	**5**	**7**	**9**	**Totals**
5	20	29	34	47	52	54	236
7	19	28	34	44	49	52	226
9	22	25	38	41	44	48	218
11	19	27	33	44	49	45	217
14	20	34	37	46	53	51	241
16	23	31	30	32	25	13	154
Totals	123	174	206	254	272	263	1292

Table 4.8: Test 2 - Unsolved Type 1 instances with the same set of instances for all the competition.

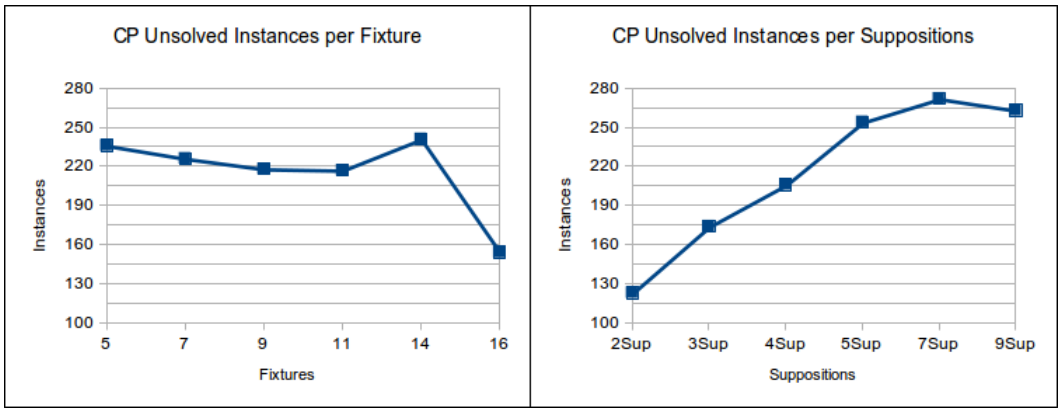

Figure 4.6: Test 2 - Unsolved Type 1 instances per fixture and number of suppositions.

As we can observe in table 4.8 and figure 4.6, this second test behaves similarly to test 1. The amount of unsolved instances tend to decrease as the tournament unfolds while it tends to increase as the number of suppositions grow.

In general we can observe in table 4.8 that SABIO didn't solve 1292 instances out of 3000, corresponding to the 43.06% of the instances. This result is pretty high, however, it is necessary to analyse the unsolved instances in more detail. For instance, it is possible to see that for 2 suppositions, there were 19 unsolved instances in fixture 7 and 23 in fixture 16, given that we used the same set of instances for both fixtures, it brings up a question: how many of these instances are there in common between both fixtures? another way to see it is, how many instances that couldn't be solved in fixture 7, couldn't be solved in fixture 16 either?

In order to analyse this kind of behaviour, we decided to select the set of unsolved queries with 2 suppositions for fixtures 5, 7, 9, 11, and 14. Then we contrasted the unsolved instances to the ones in fixture 16 to determine the amount of unsolved queries that the fixtures share as the league unfolds:

Fixture	Common	Not Common
5	4	16
7	5	14
9	9	13
11	10	9
14	16	4
16	23	0

Table 4.9: Test 2 - Amount of unsolved instances with 2 suppositions that fixtures 5, 7, 9, 11, and 14 share with 16.

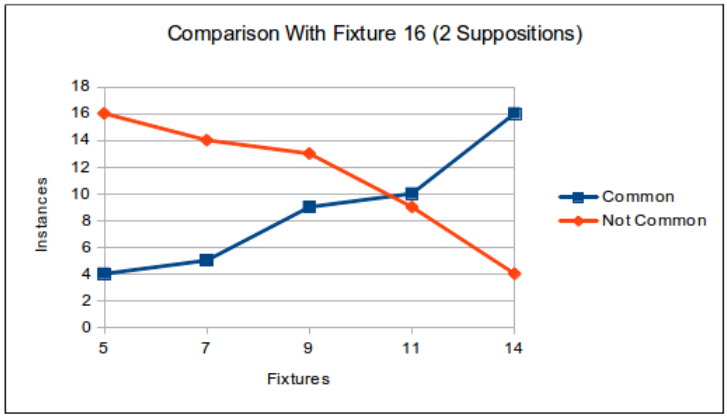

Figure 4.7: Test 2 - Amount of unsolved instances with 2 suppositions that fixtures 5, 7, 9, 11, and 14 share with 16.

Table 4.9 shows that 5 out of the 19 instances that couldn't be solved in fixture 7, could not be solved in fixture 16 either. It means that fixtures 7 and 16 have 5 instances in common that couldn't be solved. On the other hand, there were 14 instances in fixture 7 that could be solved in fixture 16.

Graphic 4.7 shows that the more distant the fixtures are to fixture 16, the lower is the the amount of instances in common. A similar behaviour can also be seen in most of the cases for the rest of the suppositions as shown in table 4.10.

Fixture	**2 Sup**	**3 Sup**	**4 Sup**	**5 Sup**	**7 Sup**	**9 Sup**
5	4	13	17	20	14	12
7	5	15	20	19	14	11
9	9	15	22	22	14	12
11	10	18	19	26	17	11
14	16	25	25	28	19	12

Table 4.10: Test 2 - Amount of unsolved instances with different suppositions that fixtures 5, 7, 9, 11, and 14 share with 16.

4.3.3. Third Set of Tests

The previous tests were configured with a time limit of 30 seconds in order to evaluate each instance for feasible solutions and trying to simulate the experience that a user "could expect" from a web application. This test pretends to analyse the behaviour of SABIO for a time limit of 10 minutes.

Test Configuration

This test was executed using only a set of 100 instances with type 1 constraints that include 2 suppositions. The instances that we used, were created for fixture 16 in test 2. We recall that the queries were evaluated in fixtures 5, 7, 9, 11, 14, and 16, in order to identify the behaviour of the application using a configuration that allows to search feasible solutions for each instance during 10 minutes.

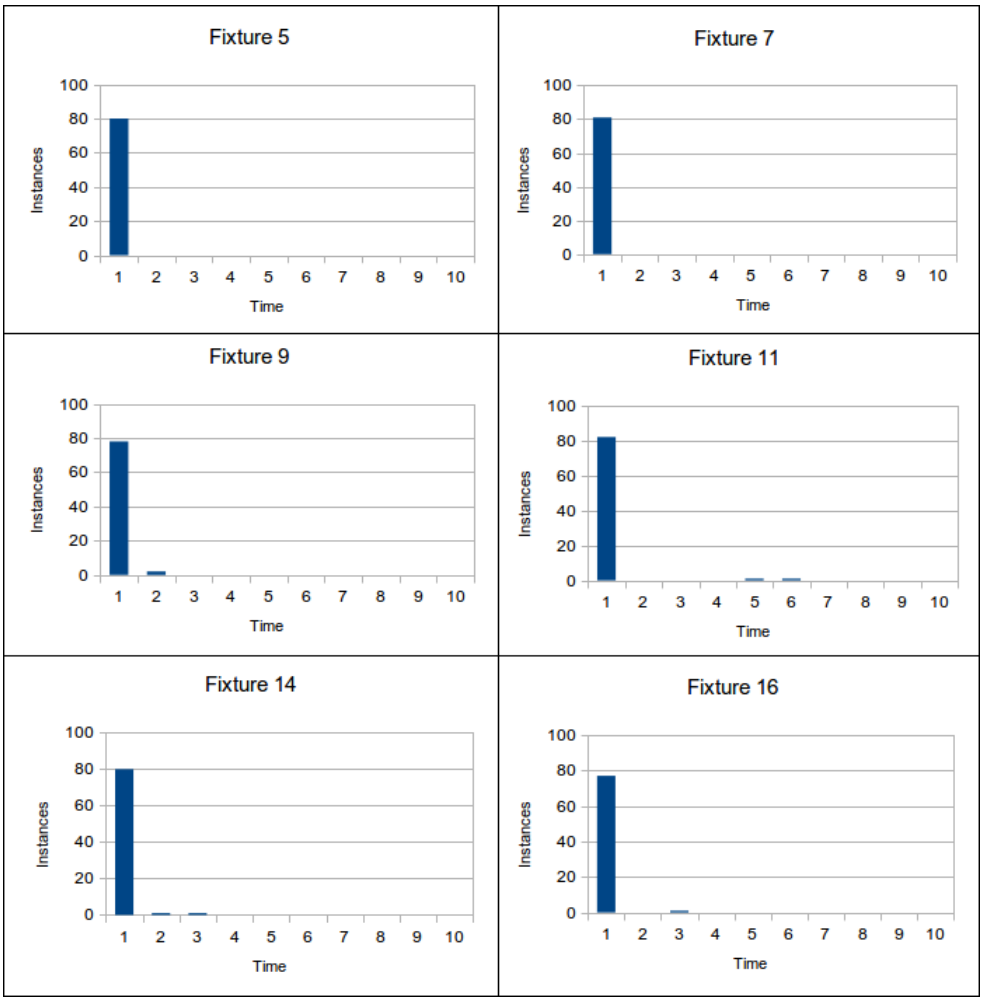

Figure 4.8: Test 3 - Amount of Type 1 instances with 2 suppositions solved within 10 minutes.

Test Results

The bar graphs from figure 4.8 show the amount of instances that could be solved within 10 minutes from a total of 100 instances per fixture. As we can observe, most of the instances were solved within the first minute, after that time, only a few amount of instances could be solved. On the other hand, the

amount of unsolved instances after 30 seconds and 10 minutes were registered in table 4.11 and we can observe that the time increment does not affect significantly the amount of solved instances within the first 30 seconds. It's easy to notice that in fixture 9, SABIO didn't solve 22 instances after 30 seconds, it means that it successfully solved 78 instances. However, after incrementing the time to 10 minutes it successfully solved 80 instances (i.e., 20 unsolved). In general, SABIO could solve only 2 more instances after incrementing the time 20 times the initial time in fixture 9. A similar behaviour can be observed in the rest of the fixtures.

Fixture	**30 Seconds**	**10 Minutes**
5	20	20
7	19	19
9	22	20
11	19	16
14	20	18
16	23	22
Totals	123	115

Table 4.11: Test 3 - Amount of unsolved instances after 30 seconds and 10 minutes.

4.4. Analysis of a Hard Family of Instances for SABIO

Every test in this section has contributed to focus our attention in a set of constraints (i.e., Type 1 constraints or position in ranking) that we will analyse in more detail in this numeral:

1. In the first set of tests, we managed to identify that SABIO presented the highest percentage of unsolved instances for constraints Type 1 and we decided to select this kind of constraints in order to run tests 2 and 3 (See section 4.3.1 for further information).

2. In the second set of tests, we observed that the same set of instances applied in different stages of the tournament, led to different solved/unsolved instances from one fixture to another. We also identified that the shorter is the distance between two fixtures, the higher is the amount of common unsolved instances between them (for more details about the test, see section 4.3.2).

3. In the third set of tests, we could identify that the highest amount of solved instances was produced within the first 30 seconds of execution. We also observed that an increment in execution time of up to 20x (i.e., 10 minutes)

does not increase the amount of solved instances significantly (See section 4.3.3 for detailed information).

4.4.1. Logical Operators Analysis of Type 1 Constraints

Recall from section 4.2.1 that a Type 1 constraint or position in ranking is used to impose constraints about the positions of the teams at the end of a competition using a logical operator ($<, >, \leq, \geq, =$), for instance, *R.Madrid will be in the same position as 1.* In order to analyse this kind of constraints, we created 20 type 1 instances using each logical operator. Then SABIO was configured with a search time limit of 30 seconds per instance and we evaluated the queries in fixtures 5, 7, 9, 11, 14, and 16. We present the obtained results in table 4.12.

Fixture	$=$	$<$	$\leq$	$>$	$\geq$
5	13	0	0	1	0
7	12	0	0	1	1
9	9	0	0	2	2
11	7	0	0	2	1
14	6	0	0	3	5
16	7	0	0	4	5
Totals	54	0	0	13	14

Table 4.12: Unsolved instances for logical operators evaluation using Type 1 constraints.

We can observe in table 4.12 that the equals operator presented the highest amount of unsolved queries, which is a expected behaviour, given that the problem of determining if any outcome of the remaining games can place team i in position x is a known NP-Complete problem. We can also observe that the amount of unsolved instances for this operator tend to decrease as the competition unfolds. Here we present an analysis of this behaviour and overview the 2 possible causes of the low performance that we managed to identify in the former version of SABIO:

1. SABIO does not have any mechanism to calculate and keep track of the partial positions while the branching algorithm assigns values to the variables, this means that the standing table is calculated only at the end, when all the variables are bounded and no inferences are made during the branching process. Therefore, if a tournament is in a fixture k and we want to check if team i that starts in position $spos_i$ can finish in position $pos_i = x$, then to check for feasibility, SABIO distributes on the pointing variables of every team based on some initial heuristics and at the end calculates the positions using a comparison mechanism implemented with boolean variables as explained in section 4.2.2.

2. As shown in section 4.2.2, every team has associated one variable per game in order to represent the points. There are cases in which the actual value selection strategies prevent from reaching certain position efficiently, this behaviour is mainly because the global value selection (min, mid, max) affects all the variables associated to a team, which makes it difficult to reach certain positions in the standing table.

Branching Heuristics Analysis of SABIO:

In this section we will analyse the branching heuristics of SABIO and show how the global strategies for value selection covered in section 4.2.2 affect the search. A soccer competition with n teams must end up with each team assigned to a position between 1 to n. If the competition still has several games to play for a particular team i, then when applying a "max" value selection strategy on its pointing variables, the team will tend to be in the first positions of the table, similarly happens when applying a "min" value selection strategy, thus placing the team at the bottom of the standing table. Keeping this behaviour in mind, we managed to represent in figure 4.9 a division of the positions that can be "easily reached"using a losing strategy for constraints like ($pos_i > x$ and $pos_i \geq x$) or a wining strategy for constraints like ($pos_i < x$ and $pos_i \leq x$):

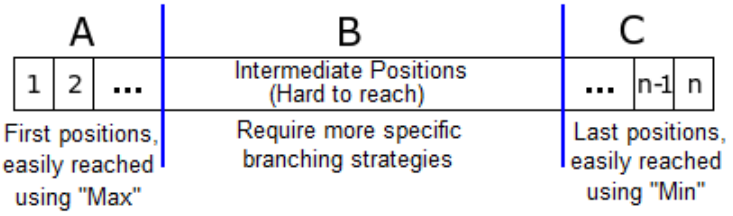

Figure 4.9: Standing table positions.

- A: Represents the top positions of the standing table that SABIO can reach using a winning strategy (i.e., max).
- B: Represents the intermediate positions that might be harder to reach with strategies like *min* or *max* given that they require more specific branching strategies.
- C: Represents the bottom positions of the standing table that SABIO can reach using a losing strategy (i.e., min).

Based on the standing table divisions depicted on table 4.9, we manage to analyse the impact of the value selection heuristics assigned to the teams variables during search depending on the logical operator ($>, \geq, <, \leq, =$) used to constraint the team position:

Case 1: ($<, \leq$) better / better or equals to: If team i starts in position $spos_i$ and we request SABIO to check if it is possible that team i finishes in a position pos_i better than x (i.e., $pos_i < x$), then what happens according to the heuristics explained in 4.2.2 is that SABIO will branch on the variables of team i first and will use "max" or a winning strategy. On the other hand, the rest of the teams will be assigned a "min" or losing strategy. This branching heuristics are generally correct, particularly when there are several games to play because it can move team i to the top positions of the standing table very quickly.

Case 2: ($>, \geq$) worse / worse or equals to: If team i starts in position $spos_i$ and we request SABIO to check if it is possible that team i finishes in a position pos_i worse than x (i.e., $pos_i > x$), then SABIO will first branch on the variables associated to team i and will use "min" or a losing strategy while the rest of the teams will be assigned a "max" or winning strategy. As in case 1, this branching heuristics are generally correct, specially when there are several games to play because it can move team i to the bottom positions of the standing table very quickly.

Case 3: ($=$) equals to: If team i is in position 9 (i.e., $spos_i = 9$) and SABIO is required to check if it's possible that team i finishes in one of the top positions (e.g., $pos_i = 1$), then the winning strategy applied to the variables of team i is very convenient. Similarly, using a losing strategy if we want SABIO to check if it's possible that team i finishes in one of the last positions (e.g, $pos_i = 20$, for a twenty team tournament). However, we have identified a series of cases when these heuristics might not be appropriate when using the equals operator to constrain a team to a position:

- Suppose that the starting position of team i is 9 (i.e., $spos_i = 9$), if we request SABIO to check if it is possible that team i finishes in position 8 (i.e., $pos_i = 8$), then a winning strategy is applied to the variables of team i, nevertheless, in order to go from position 9 to 8, team i does not necessarily have to win all the games, specially if there are several games to play. This heuristic can get team i to quickly move away from position 8 into the top positions of the standing table and it would require many backtracks from the search algorithm in order to reach such position. The same situation applies with the losing strategy applied when team i is in position 9 (i.e., $spos_i = 9$) and SABIO is required to check if it's possible that team i finishes in position 10 (i.e., $pos_i = 10$).

  If we observe figure 4.9, for the cases when a team i starts in an intermediate position $spos_i \in B$ and SABIO is required to check if it's possible that team i ends up in another intermediate position $pos_i \in B$, depending on the applied branching strategy (min or max) the team will probably overshoot the wanted place, ending up at the top of the standing table if ($pos_i < spos_i$)

or at the bottom of the standing table if ($pos_i > spos_i$).

- Suppose now that the starting position of team i is 3, if we request SABIO to check if it is possible that team i finishes in position 4 (i.e., $pos_i = 4$), then a losing strategy is applied to the variables of team i while the rest of the teams are assigned a winning strategy, however, as in the previous case, team i does not have to lose all their games in order to go from position 3 to 4 and it will overshoot the wanted place, specially if team i has several games to play.

  If we observe figure 4.9, for the cases when a team i starts in the first positions $spos_i \in A$ and SABIO is required to check if it's possible that it ends up in another first position $pos_i \in A$ where $pos_i > spos_i$, SABIO will apply a losing strategy to the variables of team i that will probably take team i to finish in the intermediate or last positions of the standing table (i.e, sections B and C).

- Finally, suppose that the starting position of team i is 17 (i.e., $spos_i = 17$), if we request SABIO to check if it is possible that team i finishes in position 16 (i.e., $pos_i = 16$), then a winning strategy is applied to the variables of team i while the rest of the teams are assigned a losing strategy. This strategies will get team i to overshoot the wanted position if i has several games to play.

  If we observe figure 4.9, for the cases when a team i starts in the last positions $spos_i \in C$ and SABIO is required to check if it's possible that it ends up in another last position $pos_i \in C$ where $pos_i < spos_i$, SABIO will apply a winning strategy to the variables of team i that will probably take team i to finish in the intermediate or first positions of the standing table (i.e, sections B and A).

4.5. Conclusions

SABIO is a constraint programming platform that can be used to tackle different soccer computational problems. For instance, the elimination problem, the score vector problem, the guaranteed qualification problem, etc. In order to represent such problems, SABIO has an interactive interface that allows users to define scenarios by using 4 kind of constraints (i.e., position in ranking, relative position, final score, and game results). SABIO uses a series of heuristics for value/variable selection based on a priority mechanism in order to look for feasible answers, however, there are some queries where the solver performs poorly and trying to configure the search algorithm might not be an accurate option for inexperienced users.

A series of tests on the former version of SABIO showed that there are two possible causes for the low performance of the solver. The first cause concerns to

the fact that SABIO does not have a mechanism to make inferences about the teams positions of the teams during search, in order to prune the search tree. The second cause is that *position in ranking constraints* (e.g., R.Madrid will be in the same position as 1) are among the hardest constraints for our model and it is worth noticing that these results are consistent with the literature [8], [9] which indicate that position in ranking queries are indeed a known NP-Complete problem.

An empirical test and a deeper analysis on the impact of the value selection heuristics during the search and its relation with the logical operators used in position in ranking constraints, allowed us to identify a family of hard instances to solve with the former version of SABIO. The test showed that when a team is constrained to a position with the equality operator, it has the greatest incidence of unsolved instances and we manage to identify 3 cases in which the predefined heuristics failed and caused what we called "position overshooting".

A mechanism to prune the search tree as the search takes place and the performance of SABIO with position in ranking constraints that use the equality operator, will constitute fundamental components to improve in the following chapter.

References

[6] L. Michel and P. V. Hentenryck, «A constraint-based architecture for local search», in *ACM SIGPLAN Notices*, ACM, vol. 37, 2002, pp. 83–100.

[7] J.-C. Régin, «Solving problems with cp: Four common pitfalls to avoid», in *Principles and Practice of Constraint Programming–CP 2011*, Springer, 2011, pp. 3–11.

[8] T. Bernholt, A. Gälich, T. Hofmeister, and N. Schmitt, «Football elimination is hard to decide under the 3-point-rule», in *MFCS*, 1999, pp. 410–418.

[9] W. Kern and D. Paulusma, «The new fifa rules are hard: Complexity aspects of sports competitions», *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 317–323, 2001.

[40] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, «F-race and iterated f-race: An overview», in *Experimental methods for the analysis of optimization algorithms*, Springer, 2010, pp. 311–336.

[61] C. Lucena, T. Noronha, S Urrutia, and C. Ribeiro, «A multi-agent framework to retrieve and publish information on qualification and elimination data in sports tournaments», *Rio de Janeiro, Brazil: Monografias em Ciência da Computação*, 2006.

[62] L. F. Vargas Rojas, «Diseño e implementación de una aplicación web y móvil para consultar sabio», Tesis de Pre-grado, 2014.

[63] F. Hutter, H. H. Hoos, and T. Stützle, «Automatic algorithm configuration based on local search», in *AAAI*, vol. 7, 2007, pp. 1152–1157.

Chapter 5

SABIO - CSP Improvements and Machine Learning

Contents

5.1. Introduction

Two of the goals of this research (i.e., *G3* and *G4* depicted in section 1.2) consist in determining the most appropriate learning techniques to improve the search algorithm in SABIO, for which it's necessary to determine an array of features and instances in order to create a data set to train a model that helps during search. On the other hand, according to the findings in the previous chapter, we suspect that SABIO presents a low performance in Type 1 queries mainly for two reasons: first, it does not have a mechanism to make inferences about the teams positions during search and second, we found a series of cases for type 1 constraints that involve the equality operator where the implemented heuristics tend to fail and teams overshoot the expected positions.

In section 5.2, we formally present the constraint programming model of SABIO and to address the first problem, in numeral 5.2.2 we propose a constraint programming model that implements a series of redundant constraints that use the positions of the teams as the branching heuristics take place, in order to prune the search tree. For the second problem, in numeral 5.2.4 we propose the creation of a classifier that evaluates instances with type 1 constraints that involve the equality operator. The classifier is used to define value selection strategies based on probabilities to win, tie or lose particular games, in order to reach the wanted position and reduce position overshooting. Finally, in section 5.3 we present an empirical comparison of the former version of SABIO and the improved model proposed here configured with a restarting mechanism for the test.

5.2. SABIO CP Model

In the following, we describe a CP formulation for soccer competitions under the 3-point-rule system. We first introduce the parameters, variables and notations used in our model, we differentiate between five categories of variables: basic formulation, game result queries, position in ranking queries, relative position queries and final point queries (See section 4.2.1 for detailed information about the constraint types in SABIO).

Basic Formulation Variables: these variables capture basic information to formulate a CP model for soccer competitions.

- n: number of teams in the competition;
- T: set of team indexes in the competition;
- i, j: team indexes, such that $(i, j \in T)$;
- p_i: initial points of team i. If i has not played any games, then $p_i = 0$;
- F: number of fixtures left to be played in the competition. A fixture consists of one or more games between competitors;
- k: represents a fixture number, $(1 \leq k \leq F)$;
- G: set that represents the schedule of the remaining games to be played. Every game is represented as a triple $ng_e = (i, j, k) \wedge 0 \leq e \leq |G|$, where k is the fixture when both teams meet in a game;
- pt_{ik}: represents the points that team i gets in fixture k, $(1 \leq k \leq F$ and $pt_{ik} \in \{0, 1, 3\})$. If team i is not scheduled to play fixture k, then $pt_{ik} = 0$.
- tp_i: total points of team i at end of the competition;

- geq_{ij}: boolean variable indicating if team j has greater or equal total points as i: if $tp_j \geq tp_i$ then $geq_{ij} = 1$; otherwise $geq_{ij} = 0 \quad (\forall i, j \in T)$;
- eq_{ij}: boolean variable indicating if two different teams i and j tie in points at the end of the competition: if $tp_j = tp_i$ and $i \neq j$ then $eq_{ij} = 1$; otherwise $eq_{ij} = 0 \quad (\forall i, j \in T)$.
- pos_i: position of team i at the end of the competition;
- $worstPos_i$: upper bound for pos_i;
- $bestPos_i$: lower bound for pos_i;

Game result queries: we use this set of variables to represent user defined assumptions of the remaining games, e.g., Barcelona ends in a tie with R. Madrid.

- Q: set of game result queries for a pair of teams (i, j) in a fixture k. Every query is defined as a tuple $nq_a = (ptc_{ik}, ptc_{jk})$ and $0 \leq a \leq |Q|$;
- ptc_{ik} and ptc_{jk}: are user suppositions about the points that a pair of teams (i, j) will get in a fixture k, i.e.,$(ptc_{ik}, ptc_{jk}) \in \{(0, 3), (3, 0), (1, 1)\}$);

Position in Ranking Queries: we use this set of variables to represent queries of teams at the end of the competition, e.g., Barcelona will be in a better position than 3.

- P: set of possible position in ranking queries, defined as a set of triples $np_b = (i, opr_i, ptn_i)$ and $0 \leq b \leq |P|$;
- opr_i: logical operator ($opr_i \in \{<, \leq, >, \geq, =\}$) to constrain team i;
- ptn_i: denoting the expected position for team i; $1 \leq ptn_i \leq n$;

Relative Position Queries: we use these variables to represent queries about relative positions between two teams, e.g., Barcelona will be in a better position than R. Madrid.

- R: set of possible relative position queries defined as a set of triples $nr_c = (i, op_{ij}, j)$ and $0 \leq c \leq |R|$;
- op_{ij}: denoting a logical operator ($op_{ij} \in \{<, \leq, >, \geq, =\}$) to constrain a pair of teams i and j.

Final Point Queries: (also known as score queries) we use these variables for queries about the final points of the teams, e.g., Barcelona scores at the end of the competition 75 points.

- S: set of possible final point queries defined as a set of tuples $ns_d = (i, s_i)$ and $0 \leq d \leq |S|$;
- s_i: denoting the wanted final points of team i.

5.2.1. CP Model Formulation

Basic Soccer Model: Constraints (5.1), (5.2), and (5.3) represent a valid game point assignment (0,3), (3,0) or (1,1) for each game $ng_e \in G$ between two teams i and j in a fixture k:

$$(0 \leq pt_{ik} \leq 3) \wedge (0 \leq pt_{jk} \leq 3) \quad \forall ng_e \in G \wedge ng_e = (i, j, k) \tag{5.1}$$

$$(pt_{ik} \neq 2) \wedge (pt_{jk} \neq 2) \quad \forall ng_e \in G \wedge ng_e = (i, j, k) \tag{5.2}$$

$$2 \leq pt_{ik} + pt_{jk} \leq 3 \quad \forall ng_e \in G \wedge ng_e = (i, j, k) \tag{5.3}$$

Constraint (5.4) corresponds to the final points tp_i of a team i. It is the addition of the initial points p_i and the points pt_{ik} obtained in every fixture k:

$$tp_i = p_i + \sum_{k=1}^{F} pt_{ik} \quad \forall i \in T \tag{5.4}$$

Constraints (5.5) to (5.8) are used to calculate final positions. All the final positions must be different and every position is bounded by $bestPos_i$ and $worstPos_i$:

$$\begin{aligned} geq_{ij} &= \begin{cases} 1, & \text{if } tp_j \geq tp_i \\ 0, & otherwise \end{cases} \quad \forall i, j \in T \\ worstPos_i &= \sum_{j=1}^{n} geq_{ij} \quad \forall i, j \in T \end{aligned} \tag{5.5}$$

$$\begin{aligned} eq_{ij} &= \begin{cases} 1, & \text{if } tp_j = tp_i \text{ and } i \neq j \\ 0, & otherwise \end{cases} \quad \forall i, j \in T \\ bestPos_i &= worstPos_i - \sum_{j=1, j \neq i}^{n} eq_{ij} \quad \forall i, j \in T \wedge i \neq j \end{aligned} \tag{5.6}$$

$$bestPos_i \leq pos_i \leq worstPos_i \quad \forall i \in T \tag{5.7}$$

$$(pos_i \neq pos_j) \quad \forall i, j \in T \wedge i \neq j \tag{5.8}$$

Game Result Queries: constraint (5.9) allows users to include assumptions of the outcome of remaining games to constrain the points of teams i and j in a particular fixture k:

$$(pt_{ik} = ptc_{ik} \wedge pt_{jk} = ptc_{jk}) \quad \forall nq_a \in Q \wedge nq_a = (ptc_{ik}, ptc_{jk}) \tag{5.9}$$

Position in Ranking Queries: involves a set of constrained teams and indicates whether a given team can be above, below, or at a given position ptn_i, constraint (5.10) depicts the five possibilities:

$$\forall np_b \in P \wedge np_b = (i, opr_i, ptn_i) \begin{cases} pos_i = ptn_i, & \text{if } opr_i \text{ is } = \\ pos_i < ptn_i, & \text{if } opr_i \text{ is } < \\ pos_i \leq ptn_i, & \text{if } opr_i \text{ is } \leq \\ pos_i > ptn_i, & \text{if } opr_i \text{ is } > \\ pos_i \geq ptn_i, & \text{if } opr_i \text{ is } \geq \end{cases} \tag{5.10}$$

Relative Position Queries: This queries indicate whether a given team i will be above, below, or equal to another team j at the end of the tournament and constraint (5.11) depicts the five queries. In this particular case we use tp_i and tp_j instead of pos_i and pos_j. We consider that two teams i and j might tied up in the same position if they have the same points at the end of the competition. We recall that we don't use pos_i and pos_j due to constraint (5.8) which indicates that two teams must have different positions at the end of the competition.

$$\forall nr_c \in R \wedge nr_c = (i, op_{ij}, j) \begin{cases} tp_i = tp_j, & \text{if } op_{ij} \text{ is } = \\ tp_i < tp_j, & \text{if } op_{ij} \text{ is } < \\ tp_i \leq tp_j, & \text{if } op_{ij} \text{ is } \leq \\ tp_i > tp_j, & \text{if } op_{ij} \text{ is } > \\ tp_i \geq tp_j, & \text{if } op_{ij} \text{ is } \geq \end{cases} \tag{5.11}$$

Final Point Queries: constraints from (5.12) guarantee a set S of *final point queries*. These total points are bounded by the initial points p_i and the maximum possible points $3 * F$ in the remaining fixtures F:

$$\begin{aligned} (tp_i = s_i) \quad & \forall ns_d \in S \wedge ns_d = (i, s_i) \\ p_i \leq s_i \leq p_i + 3 * F \quad & \forall ns_d \in S \wedge ns_d = (i, s_i) \end{aligned} \tag{5.12}$$

5.2.2. Reified Constraints for Position in Ranking Queries

In the previous chapter (4.4) we found that SABIO does not have a mechanism to make inferences about the teams positions during search, on the other hand, in CP many problems involve logical combinations of constraints that can be achieved by associating a constraint to a boolean variable that is set to 1 if the constraint holds and to 0 otherwise. This is termed a reified constraint [12].

According to the model presented in section 5.2.1, every team has a variable tp_i that represents the total points of team i at the end of the competition and equation 5.4 calculates it as the addition of the initial points p_i and the points that team i gets in every fixture k (i.e., pt_{ik}).

Suppose now that we have a competition with four teams (A, B, C, D) and we want to validate the following position in ranking constraint: *"A will be in the same position as 1"* which can be represented as the triplet $(A, =, 1)$ or $pos_A = 1$ according to equation 5.10. In the former version of SABIO, the positions are calculated after branching on the pointing variables per team and then the position constraints are validated. This configuration leads to an exhaustive search without any pruning rules based on the teams positions. For instance, the left tree in figure 5.1 depicts briefly a possible outcome for the total points of the teams A, B, C, and D after branching on their pointing variables. Looking at the first branch of the left tree, the positions of the teams based on their total points would be (B=15, A=9, C=9, D=6) or (B=15, C=9, A=9, D=6) given that A and C are tied up in points. As we can observe, neither of the standing tables satisfy the constraint $pos_A = 1$ and the red lines indicate all paths that the search algorithm explores before finding a feasible answer:

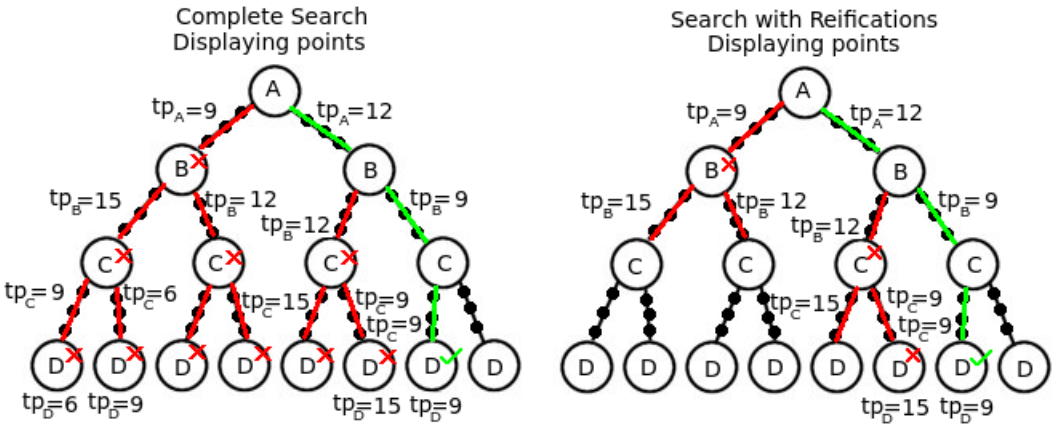

Figure 5.1: SABIO search trees with and without reification constraints.

In order to satisfy constraint $(pos_A = 1)$, *it must hold that during the search, the number of teams with more points than A has to be 0*, otherwise, A will never be in first position. It is possible to define constraints that constantly validate the number of teams with more points than A and make a branch fail when the constraint becomes unsatisfied. An example is depicted in the right tree of figure 5.1 where the branch fails once a team with more points than A is found and prevents the search algorithm to explore unnecessary nodes.

The reified constraints proposed in this section help prune the search tree by using information about the teams constrained with position in ranking queries $np_b \in P$ and $np_b = (i, opr_i, ptn_i)$ and we start by introducing 2 new variables:

- $less_{ij}$: reified variables used to compare teams final points: if $tp_j < tp_i$ then $less_{ij} = 1$; otherwise $less_{ij} = 0 \quad (\forall i, j \in T)$

- $grtr_{ij}$: reified variables used to compare teams final points: if $tp_j > tp_i$ then $grtr_{ij} = 1$; otherwise $grtr_{ij} = 0 \quad (\forall i, j \in T)$.

"Less than" operator reification: When a team i is constrained to a position ptn_i using the less than operator ($pos_i < ptn_i$), during the search, there can't be $(ptn_i - 1)$ or more teams with greater final points than team i, otherwise, the position won't be reachable.

$$grtr_{ij} = \begin{cases} 1, & \text{if } tp_j > tp_i \\ 0, & otherwise \end{cases} \quad \forall j \in T \wedge \forall np_b \in P, \text{ if } opr_i = \text{“} < \text{”} \quad \sum_{j=1, j \neq i}^{n} grtr_{ij} < (ptn_i - 1) \tag{5.13}$$

Similarly, $\sum_{j=1,j\neq i}^{n} grtr_{ij} \leq (ptn_i - 1)$ to reify queries with the operator "$\leq$".

"Greater than" operator reification: When a team i is constrained to a position ptn_i using the greater than operator ($pos_i > ptn_i$), during the search, there can't be $(n - ptn_i)$ or more teams with smaller final points than team i, otherwise, the position won't be reachable.

$$less_{ij} = \begin{cases} 1, & \text{if } tp_j < tp_i \\ 0, & otherwise \end{cases} \quad \forall j \in T \wedge \forall np_b \in P, \text{ if } opr_i = \text{“} > \text{”} \quad \sum_{j=1, j \neq i}^{n} less_{ij} < (n - ptn_i) \tag{5.14}$$

Similarly, $\sum_{j=1,j\neq i}^{n} less_{ij} \leq (n - ptn_i)$ to reify queries with the operator "$\geq$".

"Equal to" operator reification: For this operator, constraints (5.15) and (5.16) must be always satisfied during the search:

$$\sum_{j=1, j \neq i}^{n} grtr_{ij} < (ptn_i) \quad \forall np_b \in P, \text{ if } opr_i \text{ is: “} = \text{”} \tag{5.15}$$

$$\sum_{j=1, j \neq i}^{n} less_{ij} < (n - ptn_i + 1) \quad \forall np_b \in P, \text{ if } opr_i \text{ is: “} = \text{”} \tag{5.16}$$

5.2.3. Variable/Value Selection

In this section we formally describe the query based heuristics for variable/ value selection of SABIO which is one of the most important factors that influences the search efficiency. We start by first introducing a set of required variables in order to describe a priority mechanism to select first the team variables constrained in queries P and R:

- $spos_i$: starting position of team i before any branching strategy is applied;
- pri_i: denoting the priority of team i to be selected during branching, If team i does not appear in any query, then $pri_i = 0$;
- str_i: denoting the global branching strategy for the variables pt_{ik} of a particular team i in every fixture k. str_i starts with "tie" as a default value.
- x_i: variable with the number of queries in R indicating that i has to win;
- y_i: variable with the number of queries in R indicating that i has to lose.

Heuristics for Relative Position Queries: Recall from (5.11) that we use total points tp_i and tp_j to directly constrain final points. Suppose we have a query indicating $(tp_i > tp_j)$, then it's natural to try $str_i = win$ and $str_j = lose$, add up a priority of 1 to both teams ($pri_i = pri_i + 1$ and $pri_j = pri_j + 1$), and also increment the variables ($x_i = x_i + 1$ and $y_j = y_j + 1$). The heuristics for operators $(<, \leq, >, \geq)$ can then be generalized as:

- *Variable selection:* The priority of team i can be calculated as $(pri_i = x_i + y_i)$.
- *Value selection:*The branching strategy str_i can be determined by the greatest indicator between x_i and y_i or defined as $str_i = tie$ when $x_i = y_i$.

Suppose now that we have a query indicating $(tp_i = tp_j)$. Empirically, we found 2 facts about this constraint. First, it requires a high priority. Second, a (win,lose) strategy may generate final points overshooting, therefore, we decided to generalize the heuristics for (=) operator as follows:

- *Variable selection:* The priority of each team involved in the equality constraint would be $(pri_i = 10 + x_i + y_i)$.
- *Value selection:* The branching strategy for the teams involved in the constraint would be $str_i = tie$.

Heuristics for Position in Ranking Queries: Recall from (5.10) that we use the position pos_i to constrain a team to a wanted position ptn_i. Suppose we have the query $(pos_i < 8)$. It's natural to try $str_i = win$ and assign a priority according to the possible complexity of the query (i.e., the harder the position, the higher is the priority). We depict in In (5.17) the general rules for variable value selection:

$$\forall np_b \in P \wedge \; np_b = (i, opr_i, ptn_i) \begin{cases} \text{if } opr_i \text{ is } < \text{ or } \leq, & pri_i = n - ptn_i \wedge str_i = win \\ \text{if } opr_i \text{ is } > \text{ or } \geq, & pri_i = ptn \wedge str_i = lose \\ \text{if } opr_i \text{ is } =, & pri_i = ptn \wedge \begin{cases} str_i = lose, \text{ if } ptn_i > n/2 \\ str_i = win, \text{ otherwise} \end{cases} \end{cases} \tag{5.17}$$

Interestingly the defined heuristics for relative position queries R work better than the heuristics for position in ranking queries P (see section 6.3). An analysis of the logical operators (see section 4.4.1) revealed that this behavior is mostly because the defined heuristics in for queries with the "=" operator fail. For instance, every team i has some initial points p_i that determine a starting position $spos_i$. Suppose a scenario with a query ($pos_i = 7$) where $spos_i = 9$ with $F = 8$ fixtures to play. Given that the starting position is 9 and we have to reach position 7, the global branching strategy $str_i = win$ causes that pos_i overshoots position 7 and would require many backtracks of the search algorithm in order to reach such position. In the following numeral we tackle this problem by training a classifier that will determine a branching strategy based on probabilities to select among (win, tie, lose) strategies depending on the instance to solve.

5.2.4. Machine Learning for Value Selection

For teams constrained with the "=" operator in position in ranking queries, we decided to assign a high priority ($pri_i = |spos_i - ptn_i| \cdot n$) and to avoid position overshooting, we trained a classifier that selects among 9 different branching strategies as shown in table 5.1. Each strategy defines probabilities to select among (win, tie, lose), e.g, S7 means that every variable pt_{ik} of team i will be assigned *win* with a probability of 0.5, *tie* and *lose* with a probability of 0.25 each. Notice that the first 3 strategies S1, S2, and S3 correspond to the global strategies win, tie, and lose respectively:

Win probability	Tie probability	Lose probability	Strategy
1	0	0	S1
0	1	0	S2
0	0	1	S3
0.5	0.5	0	S4
0.5	0	0.5	S5
0	0.5	0.5	S6
0.5	0.25	0.25	S7
0.25	0.5	0.25	S8
0.25	0.25	0.5	S9

Table 5.1: New strategies based on probabilities.

Identifying an Array of Features: Calculating the features of a given instance shouldn't be a computationally expensive process, therefore, we managed to identify an array of 5 features for teams constrained to a position with the equals operator that might be useful for the classifier in order to decide which strategy should be applied:

- *Starting Position* ($spos_i$): This feature was selected because every team has an initial position depending on its initial points (p_i) and also because there

is a relation between the starting position and the wanted position in order to choose a heuristic for value selection.

- *Wanted Position* (ptn_i): This feature was selected because there is a relation between the starting position and the wanted position in the constraint, in order to choose a heuristic for value selection.

- *Direction and Distance* ($spos_i - ptn_i$): This feature determines the direction where a team has to move in the standing table (i.e., up or down) and also represents the distance or the number of positions that the team has to move. This feature has the following characteristics:

  - If the value of ($spos_i - ptn_i$) is positive, it indicates that the team has to go up in the standing table and the greater is the distance, the more games the team has to win.
  - If the value of ($spos_i - ptn_i$) is negative, it indicates that the team has to go down in the standing table and the smaller is the distance, the more games the team has to lose.

- *Fixtures to Play* (F): This feature might determine how fast a team has to reach certain position. For instance, if a team has few fixtures to play, it has to reach the wanted position very quickly.

- *Range Rate* ($|spos_i - ptn_i|/n$): This feature determines the rate of the distance from the starting position to the wanted position in relation to the number of teams.

Creating the Data Set: In order to create a data set, a total of 500 position in ranking queries with the equality operator were created at different stages of a tournament (fixture 7, 9, 11, 14 and 16). The tournament had 18 teams and the matches were scheduled in a single round robin. We run every query with each of the 9 branching strategies (S1 to S9) for 2 seconds in order to get the strategy that solved the instance in the shortest time. Finally, we created a data set (i.e., as an Arff file[1]), featuring the following data: starting position ($spos_i$), wanted position (ptn_i), direction and distance ($spos_i - ptn_i$), fixtures to play (F), range rate ($|spos_i - ptn_i|/n$), best executing strategy $\in \{s1, s2, s3, s4, s5, s6, s7, s8, s9\}$. An example of the .arff file with 20 instances can be seen in table 5.2.

[1]ARFF (Attribute-Relation File Format) file is an ASCII text file that describes a list of instances sharing a set of attributes. These ARFF files were developed by the Machine Learning Project at the Department of Computer Science of The University of Waikato for use with the Weka machine learning software [64].

```
@RELATION SABIO

@ATTRIBUTE Direction     NUMERIC
@ATTRIBUTE StartingPos   NUMERIC
@ATTRIBUTE WantedPos     NUMERIC
@ATTRIBUTE RangeRate     NUMERIC
@ATTRIBUTE Fixtures      NUMERIC
@ATTRIBUTE Strategy      {S1,S2,S3,S4,S5,S6,S7,S8,S9}

@DATA
-1,4,5,0.0555555556,11,S8
-5,11,16,0.2777777778,11,S2
3,14,11,0.1666666667,11,S2
-13,1,14,0.7222222222,11,S3
15,18,3,0.8333333333,11,S7
5,16,11,0.2777777778,11,S8
5,7,2,0.2777777778,11,S8
-1,11,12,0.0555555556,11,S2
-6,5,11,0.3333333333,11,S6
6,10,4,0.3333333333,11,S5
3,14,11,0.1666666667,11,S2
4,5,1,0.2222222222,11,S8
2,7,5,0.1111111111,11,S8
10,13,3,0.5555555556,11,S5
9,10,1,0.5,11,S1
15,16,1,0.8333333333,11,S4
12,17,5,0.6666666667,11,S5
5,18,13,0.2777777778,11,S5
-3,7,10,0.1666666667,11,S9
11,16,5,0.6111111111,11,S4
```

Table 5.2: Data set with 20 records for instances with constraints type 1.

Training the classifier: The objective of training this classifier was that given a Type 1 query with the equality operator "=", the classifier selects one of the 9 branching strategies based on the best execution times. We used Weka (V3.6.12) and configured it with the J48 decision tree [65], then the model was trained using Cross-validation and it correctly classified 81.8% of the instances and had a relative absolute error of 27.3%.

5.3. Tests on the New Model

In order to compare the new model of SABIO proposed in this section to the former version, we decided to run the test on the same instances that we run in section 4.3.1 over 3 different type of constraints, namely, Type 1 or position in ranking, Type 2 or relative position and Type 3 or final score.

Due to the results obtained in section 4.3.1, particularly after analysing that most of the instances were solved under a time limit of 0.51 seconds on average, we decided to implement a *restaring mechanism* in SABIO in order to execute each instance from different restart points (i.e, assign different value selection strategies to unconstrained teams). This restart mechanism helped the search algorithm to scape from branches in the search tree where no solutions are found for certain time, thus having the effect of a kick move in order to explore other branches of the tree during search.

Each instance was configured to use a search time limit of 30 seconds with 10 restarts points, that is, 3 seconds to search for feasible answers in each restart. Given that the test had certain non-determinism due to the characteristics of value selection using probabilities plus the restart mechanism, we run the test twice and registered the average results in table 5.3 where we refer to the former version of SABIO as *CP* and the new version proposed in this section as *CP-ML* given that one of the improvements implements a machine learning technique.

		Type 1 Queries						Type 2 Queries						Type 3 Queries					
Fixture	Test \ Sup.	2	3	4	5	7	9	2	3	4	5	7	9	2	3	4	5	7	9
Fixture 7	CP	24	28	38	46	50	52	-	-	-	-	-	-	-	2	-	-	6	3
	CP-ML	3	4	3	5	11	17	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 9	CP	23	28	35	46	49	44	-	-	-	-	-	-	-	-	-	-	1	1
	CP-ML	1	1	4	5	10	14	-	-	-	-	-	-	-	-	-	-	-	1
Fixture 11	CP	25	27	31	44	49	47	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML	2	3	4	8	16	16	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 14	CP	23	33	38	41	51	45	-	-	-	-	-	-	-	-	-	1	-	-
	CP-ML	2	3	7	9	14	9	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 16	CP	23	31	29	31	25	13	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Unsolved Instances	CP	1069						-						14					
	CP-ML	171						-						1					
Fixture 7	CP	**.22**	**.48**	**.31**	**.41**	**.77**	**1.10**	.07	.07	.07	.07	.07	.06	.07	**.06**	.06	.06	**.22**	.22
	CP-ML	.59	.63	1.04	1.05	.95	1.19	**.06**	**.06**	**.06**	**.06**	**.06**	.06	**.06**	.12	**.05**	.06	.26	.22
Fixture 9	CP	**.08**	**.41**	**.23**	**.65**	**.31**	.72	.06	.06	.06	.06	.06	.06	.06	.05	.05	.05	**.05**	.22
	CP-ML	.80	.56	1.15	1.15	1.89	**.46**	.06	.06	.06	.06	.06	**.05**	.06	.05	.05	.05	.08	**.08**
Fixture 11	CP	.56	**.10**	.75	.69	.86	.45	.06	.05	.05	.05	.05	.05	.05	.05	.07	.05	.05	.04
	CP-ML	**.40**	.39	.75	**.63**	**.77**	**.22**	.06	.05	.05	.05	.05	.05	.05	.05	.07	**.04**	**.04**	.04
Fixture 14	CP	.42	**.13**	.49	1.01	1.53	.74	.05	.04	.04	.04	.04	.04	.04	.04	.04	**.04**	.04	.04
	CP-ML	**.07**	.36	**.11**	**.16**	**.43**	**.54**	**.04**	.04	.04	.04	.04	**.03**	.04	.04	**.03**	.06	**.03**	**.03**
Fixture 16	CP	.12	.04	.89	.55	.45	.65	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04
	CP-ML	**.04**	.04	**.03**	**.06**	**.04**	**.05**	.04	**.03**	**.03**	**.03**	**.03**	**.03**	.04	**.03**	**.03**	**.03**	**.03**	**.03**
Average Times	CP	0.51 sec						0.06 sec						0.07 sec					
	CP-ML	0.55 sec						0.05 sec						0.06 sec					

Table 5.3: Unsolved instances and average running times of the former version of SABIO (CP) and the proposed model (CP with ML).

5.4. Conclusions and Result Analysis

We managed to improve SABIO's performance and in general we observed that our CP-ML model performs better that the former CP version. We could reduce the amount of unsolved instances from 1069 to 171 in type 1 queries, which represents an improvement from 64.35% to 94.3% solved instances. However, it required a lot of work and tuning given that: first, we implemented constraints to prune the search tree, second, we also created a classifier to define value selection strategies based on probabilities for position in ranking queries that used the equality operator and third, we configured the search with a restarting mechanism.

Interestingly, CP-ML also shows to be more effective in type 3 queries where it reduced the unsolved instances from 14 to 1. We attribute this behaviour to the restarting mechanism, since this kind of constraints are not affected by the proposed constraints nor by the classifier. Additionally, the differences on average time are not significant and despite the fact CP solves less instances, it seems to be faster than CP-ML in type 1 queries on average, while CP-ML seems to be faster that CP in type 2 and 3 queries.

References

[6] L. Michel and P. V. Hentenryck, «A constraint-based architecture for local search», in *ACM SIGPLAN Notices*, ACM, vol. 37, 2002, pp. 83–100.

[7] J.-C. Régin, «Solving problems with cp: Four common pitfalls to avoid», in *Principles and Practice of Constraint Programming–CP 2011*, Springer, 2011, pp. 3–11.

[8] T. Bernholt, A. Gälich, T. Hofmeister, and N. Schmitt, «Football elimination is hard to decide under the 3-point-rule», in *MFCS*, 1999, pp. 410–418.

[9] W. Kern and D. Paulusma, «The new fifa rules are hard: Complexity aspects of sports competitions», *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 317–323, 2001.

[40] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, «F-race and iterated f-race: An overview», in *Experimental methods for the analysis of optimization algorithms*, Springer, 2010, pp. 311–336.

[61] C. Lucena, T. Noronha, S Urrutia, and C. Ribeiro, «A multi-agent framework to retrieve and publish information on qualification and elimination data in sports tournaments», *Rio de Janeiro, Brazil: Monografias em Ciência da Computação*, 2006.

[62] L. F. Vargas Rojas, «Diseño e implementación de una aplicación web y móvil para consultar sabio», Tesis de Pre-grado, 2014.

[63] F. Hutter, H. H. Hoos, and T. Stützle, «Automatic algorithm configuration based on local search», in *AAAI*, vol. 7, 2007, pp. 1152–1157.

Chapter 6

SABIO as a MIP Application for Soccer Analysis

Contents

6.1. Introduction

In this section, we describe a MIP formulation for soccer competitions and first we describe the variables and notations used in our model. As in the CP and CP-ML models from the previous chapters, we differentiate between five categories of variables: basic formulation, game result queries, position in ranking queries, relative position queries and final point queries.

The basic formulation will serve as a platform to define the soccer competition pointing rules, in order to represent the matches among the teams and thus calculate their final points to determine their positions. The later parts of the model, to which we will refer as "queries", will let the users add soccer assumptions in form of linear equations. In order to create the basic formulation, some ideas from general literature were kept in mind, for instance, the usage of a set of variables g_{ij} representing remaining games, first proposed by [53], [55] to model the baseball elimination problem and also some of the ideas from [59] for the team point assignment under the 3-point-rule system (i.e., zero points to a loss, one point to a tie and three points to a win). However, many variables had to be added to formulate certain features of our model, e.g., flexibility of single and double

round-robin competition and variables to represent the user-defined queries.

6.2. MIP Model Variables, Parameters and Notation

Basic Formulation Variables: these variables capture basic information to formulate a model for soccer competitions.

- n: number of teams in the competition;
- T: set of team indexes in the competition;
- i, j: team indexes, such that $(i, j \in T)$;
- p_i: initial points of team i. If i has not played any games, then $p_i = 0$;
- g_{ij}: number of remaining games between a pair of teams i and j;
- w_{ij}: number of games that team i wins over team j ;
- t_{ij}: number of games that team i ties with team j ;
- l_{ij}: number of games that team i loses over team j;
- tp_i: total points of team i at end of the competition;
- pos_i: position of team i at the end of the competition;
- $worstPos_i$: upper bound for pos_i;
- $bestPos_i$: lower bound for pos_i;

Game Result Queries: we use this set of variables to represent user defined assumptions of the remaining games, e.g., Barcelona ends in a tie with R. Madrid.

- Q: set of game result queries for pairs of teams (i, j). Q is defined as a set of triples $nq_a = (wc_{ij}, tc_{ij}, lc_{ij})$ and $0 \leq a \leq |Q|$;
- wc_{ij}: minimum number of games that team i wins over team j;
- tc_{ij}: minimum number of games that team i ties with team j;
- lc_{ij}: minimum number of games that team i loses over team j;

Position in Ranking Queries: we use this set of variables to represent queries of teams at the end of the competition.

- P: set of possible position in ranking queries, defined as a set of triples $np_b = (i, opr_i, ptn_i)$ and $0 \leq b \leq |P|$;
- opr_i: logical operator ($opr_i \in \{<, \leq, >, \geq, =\}$) to constrain team i;
- ptn_i: denoting the expected position for team i; $1 \leq ptn_i \leq n$;
- L: denoting the set of team indexes included in all the triples $np_b \in P$ such that $np_b = (i, opr_i, ptn_i)$ and $i \in L$ and $L \subseteq T$;
- geq_{ij}: boolean variable indicating if team j has greater or equal total points as i: if $tp_j \geq tp_i$ then $geq_{ij} = 1$; otherwise $geq_{ij} = 0 \quad (\forall i \in L, \forall j \in T)$;
- eq_{ij}: boolean variable indicating if two different teams i and j tie in points at the end of the competition: if $tp_j = tp_i$ and $i \neq j$ then $eq_{ij} = 1$; otherwise $eq_{ij} = 0 \quad (\forall i \in L, \forall j \in T)$.

Relative Position Queries: we use these variables to represent queries about relative positions between two teams, e.g., Barcelona will be in a better position than R. Madrid.

- R: set of possible relative position queries defined as a set of triples $nr_c = (i, op_{ij}, j)$ and $0 \leq c \leq |R|$;
- op_{ij}: denoting a logical operator ($op_{ij} \in \{<, \leq, >, \geq, =\}$) to constrain a pair of teams i and j.

Final Point Queries: (also known as score queries) we use these variables for queries about the final points of the teams, e.g., Barcelona scores at the end of the competition 75 points.

- S: set of possible final point queries defined as a set of tuples $ns_d = (i, s_i)$ and $0 \leq d \leq |S|$;
- s_i: denoting the wanted final points of team i.

6.2.1. MIP Model Formulation

Basic Soccer Model: Constraints (6.1), (6.2), and (6.3) describe a basic soccer model with a valid (win, tie, lose) assignment for every game between a pair of teams (i, j) with g_{ij} games left to play. We recall that g_{ij} must be equal to g_{ji}, and it can be deduced that w_{ij} corresponds not only to the number of games that team i wins over team j, it also represents the games that team j loses over team i, then $w_{ij} = l_{ji}$. Respectively, $t_{ij} = t_{ji}$, and $l_{ij} = w_{ji}$ [59]. In this scenario, a team can get (3 points – win, 1 point – tie, 0 points – loss) in every game, then the total number of points of team i can be calculated by adding its initial points p_i

and the points obtained against every other team. Depending on the competition and the current state of the tournament g_{ij} is set to 0 (no games left between the teams), 1 or 2. Unlike previous work in [59] which is limited to single round-robin competitions, this representation is flexible to represent both single and double round-robin competitions.

$$w_{ij} + t_{ij} + l_{ij} = g_{ij} \quad \forall_{i,j \in T} \wedge g_{ij} \geq 0 \tag{6.1}$$

$$w_{ij} = l_{ji} \wedge t_{ij} = t_{ji} \wedge l_{ij} = w_{ji} \quad \forall_{i,j \in T} \tag{6.2}$$

$$tp_i = p_i + \sum_{j=1, j \neq i}^{n} 3 \cdot w_{ij} + t_{ij} \quad \forall_{i,j \in T} \tag{6.3}$$

Game Results Queries: our model will allow users to include assumptions of the outcome of remaining games in the competition, e.g., team i wins over team j in at least one of the two remaining games between the two teams. Taking this into account we extend our basic model with constraints (6.4).

$$\begin{gathered} (wc_{ij} + tc_{ij} + lc_{ij} \leq g_{ij}) \;\; \forall nq_a \in Q \wedge nq_a = (wc_{ij}, tc_{ij}, lc_{ij}) \\ (w_{ij} \geq wc_{ij}) \wedge (t_{ij} \geq tc_{ij}) \wedge (l_{ij} \geq lc_{ij}) \;\; \forall nq_a \in Q \wedge nq_a = (wc_{ij}, tc_{ij}, lc_{ij}) \\ wc_{ij}, tc_{ij}, lc_{ij} \geq 0 \end{gathered} \tag{6.4}$$

Position in Ranking Queries: A position in ranking query involves a set of constrained teams $L \subseteq T$ and indicates whether a given team can be above, below, or at a given position ptn_i, constraint (6.5) depicts the five possibilities:

$$\forall np_b \in P \wedge np_b = (i, opr_i, ptn_i) \begin{cases} pos_i = ptn_i, & \text{if } opr_i \text{ is } = \\ pos_i \leq ptn_i - 1, & \text{if } opr_i \text{ is } < \\ pos_i \leq ptn_i, & \text{if } opr_i \text{ is } \leq \\ pos_i \geq ptn_i + 1, & \text{if } opr_i \text{ is } > \\ pos_i \geq ptn_i, & \text{if } opr_i \text{ is } \geq \end{cases} \tag{6.5}$$

Constraint (6.6) indicates the number of teams j that finish the competition with better or equal points as team i. This number (i.e., $worstPos_i$) is the upper bound for pos_i as expressed in constraint (6.8).

$$\begin{gathered} geq_{ij} = \begin{cases} 1, & \text{if } tp_j \geq tp_i \\ 0, & otherwise \end{cases} \quad \forall j \in T \wedge \forall i \in L \\ worstPos_i = \sum_{j=1}^{n} geq_{ij} \quad \forall j \in T \wedge \forall i \in L \end{gathered} \tag{6.6}$$

Constraint (6.7) indicates whether teams i and j ($i \neq j$) end up with the same points at the end of the competition. The lower bound for pos_i (i.e., $bestPos_i$) can be computed by subtracting from the upper bound, the total teams in the same position as team i.

$$\begin{aligned} eq_{ij} &= \begin{cases} 1, & \text{if } tp_j = tp_i \text{ and } i \neq j \\ 0, & otherwise \end{cases} \quad \forall j \in T \wedge \forall i \in L \\ bestPos_i &= worstPos_i - \sum_{j=1, j \neq i}^{n} eq_{ij} \quad \forall j \in T \wedge \forall i \in L \end{aligned} \tag{6.7}$$

Finally, constraint (6.8) sets the bounds for the position of team i and constraint (6.9) indicates that the positions for two teams must be different.

$$bestPos_i \leq pos_i \leq worstPos_i \quad \forall i \in L \tag{6.8}$$

$$(pos_i \neq pos_j) \quad \forall i, j \in L \wedge i \neq j \tag{6.9}$$

Relative Position Queries: this queries indicate whether a given team i will be above, below, or equal to another team j at the end of the tournament and constraint (6.10) depicts the five queries. In this particular case we use tp_i and tp_j. We consider that two teams i and j might tie up in the same position if they have the same points at the end of the competition. We recall that we don't use pos_i and pos_j due to constraint (6.9) which indicates that two teams must have different positions at the end of the competition.

$$\forall nr_c \in R \wedge nr_c = (i, op_{ij}, j) \begin{cases} tp_i = tp_j, & \text{if } op_{ij} \text{ is } = \\ tp_i \leq tp_j - 1, & \text{if } op_{ij} \text{ is } < \\ tp_i \leq tp_j, & \text{if } op_{ij} \text{ is } \leq \\ tp_i \geq tp_j + 1, & \text{if } op_{ij} \text{ is } > \\ tp_i \geq tp_j, & \text{if } op_{ij} \text{ is } \geq \end{cases} \tag{6.10}$$

Final Point Queries: To guarantee a set S of *final point queries*, the model should include the linear constraints from (6.11). Notice that the score s_i should have a lower bound of p_i and an upper bound of all the possible points if team i wins all its the remaining games g_{ij}:

$$\begin{aligned} (tp_i = s_i) \quad & \forall ns_d \in S \wedge ns_d = (i, s_i) \\ p_i \leq s_i \leq p_i + 3 \cdot \sum_{j=1, j \neq i}^{n} g_{ij} \quad & \forall ns_d \in S \wedge ns_d = (i, s_i) \end{aligned} \tag{6.11}$$

Objective Function: We are tackling only decision problems; therefore, it is only necessary to find an assignment for the variables that satisfies the constraints. However, users might be interested in either minimizing or maximizing the total points for a given team as depicted in (6.12):

$$\text{maximize: } tp_i \quad (i \in T) \tag{6.12}$$

6.3. Model Comparison (CP, CP-ML and MIP)

In this section we show an empirical evaluation of our models using CPLEX (V12.6.2) as our MIP reference solver and Mozart-Oz (V 1.4.0) as our CP reference solver. In order to compare the proposed models of SABIO (i.e., CP-ML and MIP) with the former version (CP), we used CPLEX (V12.6.2) as our MIP reference solver and Mozart-Oz (V 1.4.0) as our CP reference solver. As the previous experiments, this test was executed in a 4-core machine featuring an Intel Core i5 processor at 2.3 Ghz and 4096 MB of RAM.

This test was based in a Colombian league (liga Postobón 2014-I) with 18 teams and 18 fixtures to play in a single round-robin schedule. We evaluated the models on instances previously created in section 4.3.1. The instances were created for different stages of the competition (i.e., fixtures 7, 9, 11, 14, and 16). For each fixture we created instances with position in ranking queries (Type 1), relative position queries (Type 2), and final point queries (Type 3).

Every instance was executed for a maximum time of 30 seconds, and in order to analyse the performance of the models, we experimented separately the following scenarios: the MIP implementation, the CP implementation without the machine learning classifier, the CP-ML implementation with the machine learning classifier, and a sequential execution of CP-ML and the MIP implementation.

Table 6.1 shows the number of unsolved instances (top) and the average runtime (bottom) of solved instances in the experiments for each model. In this table we observe that position ranking queries (Type 1) are the hardest ones for all models. Unlike Type 2 and Type 3 queries where nearly all instances could be solved. For position in ranking queries (Type 1) we observe an important number of unsolved instances and the highest average runtime. We attribute this to the fact, that position bounds (i.e., $bestPos_i$ and $worstPos_i$) can only be computed after finding the total points (tp_i) for the teams in the competition. Therefore, the search algorithm can only prune the domain of the variables after computing the total points for the teams. As a result, standing positions are calculated every time the search algorithm performs a complete game points assignment for all teams. It is worth noticing that these results are consistent with the literature [8], [9] which indicates that position ranking queries are indeed a known NP-Complete problem.

		Type 1 Queries						Type 2 Queries						Type 3 Queries					
Fixture	Test \ Sup.	2	3	4	5	7	9	2	3	4	5	7	9	2	3	4	5	7	9
Fixture 7	MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CP	24	28	38	46	50	52	-	-	-	-	-	-	-	2	-	-	6	3
	CP-ML	3	4	3	5	11	17	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML&MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 9	MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CP	23	28	35	46	49	44	-	-	-	-	-	-	-	-	-	-	1	1
	CP-ML	1	1	4	5	10	14	-	-	-	-	-	-	-	-	-	-	-	1
	CP-ML&MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 11	MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CP	25	27	31	44	49	47	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML	2	3	4	8	16	16	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML&MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 14	MIP	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-
	CP	23	33	38	41	51	45	-	-	-	-	-	-	-	-	-	1	-	-
	CP-ML	2	3	7	9	14	9	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML&MIP	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-
Fixture 16	MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CP	23	31	29	31	25	13	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
	CP-ML&MIP	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Unsolved Instances	MIP	1						-						-					
	CP	1069						-						14					
	CP-ML	171						-						1					
	CP-ML&MIP	1						-						-					
Fixture 7	MIP	.50	.49	.53	.65	.88	1.31	.42	.42	.42	.42	.45	.48	.43	.45	.44	.41	.43	.40
	CP	.22	.48	.31	.41	.77	1.10	.07	.07	.07	.07	.07	.06	.07	**.06**	.06	.06	.22	.22
	CP-ML	.59	.63	1.04	1.05	.95	1.19	.06	.06	.06	.06	.06	.06	.06	.12	**.05**	.06	.26	.22
	CP-ML&MIP	**.12**	**.13**	**.15**	**.19**	**.23**	**.40**	**.05**	**.05**	**.05**	**.05**	**.05**	**.05**	**.05**	**.06**	**.05**	**.05**	**.05**	**.06**
Fixture 9	MIP	.48	.46	.51	.58	.78	1.01	.41	.41	.44	.44	.42	.43	.41	.41	.44	.44	.42	.43
	CP	**.08**	.41	.23	.65	.31	.72	**.06**	.06	.06	.06	.06	.06	.06	**.05**	.05	.05	.05	.22
	CP-ML	.80	.56	1.15	1.15	1.89	.46	**.06**	.06	.06	.06	.06	.05	.06	**.05**	.05	.05	.08	.08
	CP-ML&MIP	.14	**.08**	**.16**	**.18**	**.28**	**.26**	**.06**	**.05**	**.04**	**.05**	**.04**	**.04**	**.05**	**.05**	**.04**	**.03**	**.03**	**.04**
Fixture 11	MIP	.49	.53	.51	.61	.74	1.16	.41	.42	.42	.42	.41	.40	.44	.40	.41	.44	.40	.40
	CP	.56	**.10**	.75	.69	.86	.45	.06	.05	.05	.05	.05	.05	.05	.05	.07	.05	.05	.04
	CP-ML	.40	.39	.75	.63	.77	**.22**	.06	.05	.05	.05	.05	.05	.05	.05	.07	.04	.04	.04
	CP-ML&MIP	**.09**	.14	**.15**	**.18**	**.36**	.32	**.03**	**.03**	**.04**	**.03**	**.03**	**.03**	**.03**	**.03**	**.03**	**.03**	**.03**	**.03**
Fixture 14	MIP	.43	.43	.42	.50	.60	.62	.38	.38	.41	.40	.38	.36	.40	.40	.38	.37	.38	.33
	CP	.42	.13	.49	1.01	1.53	.74	.05	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04
	CP-ML	.07	.36	.11	.16	.43	.54	**.04**	.04	.04	.04	.04	.03	.04	.04	**.03**	.06	.03	**.03**
	CP-ML&MIP	**.05**	**.06**	**.10**	**.15**	**.24**	**.26**	**.04**	**.03**	**.03**	**.03**	**.03**	**.02**	**.03**	**.02**	**.03**	**.03**	**.02**	**.03**
Fixture 16	MIP	.36	.37	.38	.39	.40	.38	.34	.33	.33	.34	.32	.35	.40	.38	.36	.39	.37	.33
	CP	.12	.04	.89	.55	.45	.65	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04	.04
	CP-ML	.04	.04	**.03**	.06	.04	.05	.04	.03	.03	.03	.03	.03	.04	**.03**	**.03**	.03	.03	.03
	CP-ML&MIP	**.02**	**.03**	**.03**	**.03**	**.03**	**.02**	**.02**	**.02**	**.02**	**.02**	**.02**	**.02**	**.02**	**.03**	**.03**	**.02**	**.02**	**.02**
Average Times	MIP	0.58 sec						0.40 sec						0.41 sec					
	CP	0.51 sec						0.06 sec						0.07 sec					
	CP-ML	0.55 sec						0.05 sec						0.06 sec					
	CP-ML&MIP	**0.16** sec						**0.04** sec						**0.04** sec					

Table 6.1: Unsolved instances and average running times of three models (MIP, CP, CP-ML) and a sequential execution (CP-ML&MIP)

In general, we have observed that the MIP model provides very robust solutions for all queries and only 1 out 9000 queries cannot be solved within the time limit. Our CP model seems to perform well for Type 2 and Type 3 queries with only 14 unsolved instances. After a detailed analyses we observed that our variable/value selection heuristics for the CP model struggles with Type 1 queries related to the "=" operator, e.g., Barcelona will finish 7 in the competition. In total we observed that 926 out of 1905 queries that include the "=" operation timed out.

To overcome the limitation of the CP model with Type 1 queries, we suggested to extend our model with a machine learning approach to identify the best branch-

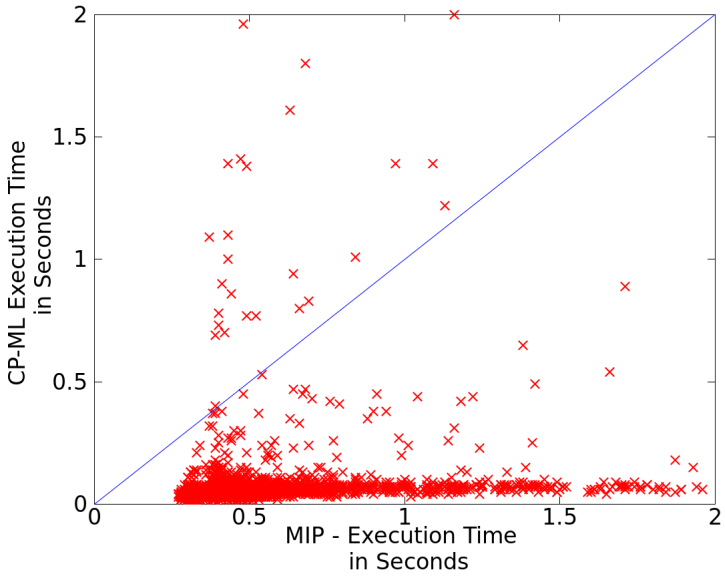

Figure 6.1: Execution times of 8551 common instances solved under two seconds

ing rule described in Section 5.2. Interestingly, the use of a machine classifier plus a restarting mechanism, considerably improves the performance of the algorithm by reducing the number of unsolved instances from 1069 to 171.

Additionally, we observed that the MIP and the CP approaches are complementary. Broadly speaking, the MIP model is able to solve more instances than the CP one, i.e., 1082 more than the pure CP approach and 171 more than the CP with machine learning. Alternatively, the CP approach is about 12%, 85%, and 85% faster than the MIP approach for queries Type 1, 2, and 3 respectively.

Figure 6.1 further compares the CP-ML and the MIP approaches showing the MIP vs. CP-ML runtime for all queries commonly solved by the two models within 2 seconds. Points below (resp. above) the diagonal indicates that the CP-ML model (resp. MIP) is better. As it can be observed the CP-ML model is typically faster than the MIP model. In fact, for 8549 out of 8829 instances CP-ML is faster than MIP.

Considering the complementary behavior of MIP and CP-ML, we decided to devise a mixed approach (i.e., CP-ML&MIP) where we execute the CP solver with a 0.5-second time limit and the MIP solver with the remaining time. Certainly, this approach exploits the strengths of both alternatives in a mixed solver. The mixed solver is about 72% faster than the pure MIP one, and 68% faster than the pure CP one on average for the overall solved instances with Type 1 constraints.

References

[12] M. Milano and M. Wallace, «Integrating operations research in constraint programming», *Annals of Operations Research*, vol. 175, no. 1, pp. 37–76, 2010.

[64] U. of Waikato. (2008). Attribute-relation file format (arff). Accessed: 2015-12-05, [Online]. Available: `http://www.cs.waikato.ac.nz/ml/weka/arff.html`.

[65] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, Third. Morgan Kaufmann, 2005, pp. 410–427.

Chapter 7

Conclusions and Future Work

Contents

In this chapter we present the main challenges of this research and we also summarize the addressed goals. For each goal, we present our contributions as well as our conclusions. At the end of the chapter, we present a discussion of the further research opportunities based on the current work.

7.1. Addressed Goals and Conclusions

We recall from previous sections that this research focused in SABIO, a constraint programming application for soccer analysis that has shown low performance in certain type of instances. Constraint programming (i.e., CP) is a paradigm that has developed several techniques to address combinatorial problems and in the past years, a growing interest in combining CP and Machine learning has been studied in order to improve CP efficiency when searching for feasible solutions.

Efficiency improvement is a general concern for the research community and the software development industry. One of the main contributions of this research is that we establish a reference point for the use of learning techniques in real world applications created under the constraint satisfaction paradigm, given that the main goal that we pretended to achieve during this research was to *implement learning techniques in the search algorithm of SABIO to improve its capacity to answer a wider range of instances by studying aspects related to machine learning in constraint programming.*

As a natural process of this kind of research, we managed to establish a series

of smaller goals that let us complete our main objective. Here we present our contributions and the corresponding conclusions based on 4 goals:

Goal 1: To create a state of the art related to machine learning in constraint programming. In chapter 2 we provided a summary of the most relevant concepts of constraint programming, machine learning and integer programming that concerned this dissertation and we determined that many approaches related to machine learning and combinatorial problems have been studied in order to improve algorithms performance for CSPs, SAT and QBF (e.g., algorithm portfolios, per-class learning and adaptive algorithms). We also found that one of the main usages of machine learning in CP problems is to improve efficiency and quality of constraints by using experience.

We found very convenient to define and identify the kind of problems that could be tackled with SABIO, then in chapter 3 we provided a state of the art related to common soccer computational problems, in order to establish a reference point of previous researches and approaches. We found that many of the problems that can be addressed with SABIO (e.g., the elimination problem, the score vector problem, etc.) are known NP-Complete problems for the actual 3 point rule system (3 points – win, 1 point – tie, 0 points – loss). While researching about computational problems related to soccer, we found that previous approaches were proposed using linear programming models. For instance, *RIOT* (Berkeley Remote Interactive Optimization Testbed), used to broadcast optimization-based information of the Major League Baseball(MLB) [54] and *FUTMAX* used to provide detailed information of the conditions of qualification and elimination of of Brazilian national soccer [59][61]. However, to the best of our knowledge, SABIO is the first interactive platform with a constraint programming approach to tackle different soccer computational problems featuring scenarios defined by the users.

Goal 2: To identify family of instances where SABIO presents low performance and that can be used to train and validate the model. In chapter 4 we performed a series of empirical tests to the former CP version of SABIO and managed to identify a family of instances where SABIO presented low performance. Particularly, the set of hard instances correspond to *position in ranking constraints* (e.g., R.Madrid will be in the same position as 1). This result about family of hard instances is consistent with the literature [8][9] since this kind of constraints are indeed known to be NP-Complete. However, we also carried out a deeper analysis on the impact of the variable/value selection heuristics of SABIO during search and its relation with the logical operators used in position in ranking constraints. This analysis let us determine that there were 2 possible reasons for such low performance. First, when a team is constrained to a position with the equality operator, the implemented global heuristics of SABIO for value selection (i.e., win, tie, lose or max, mid, min respectively) caused position overshooting (i.e., a team overpasses the wanted position) and we manage to identify 3 cases when

the heuristics might fail. Second, we also determined that SABIO did not have a mechanism to make inferences about the positions of the teams during search, which can be really important in order to prune the search tree. The identified family of instances and the two possible reasons for low performance, became important challenges that we managed to study during this research.

Goal 3: To determine the most appropriate learning techniques to improve the search algorithm of SABIO according to the actual characteristics of the application.

Goal 4: To determine the features and instances that will be used to train and validate the model by using performance tests.

In order to complete these two goals, in chapter 5 we formalize the computational model of SABIO and proposed two improvements based on the findings for goal 2. First, in order to improve the cases in which the heuristics of SABIO caused position overshooting, we managed to train a classifier that extends the branching heuristics for position in ranking queries with the equality operator where we previously identified some problems. Second, we also proposed a series of reified constraints in order improve pruning during search that uses the teams final points in order to make some reasoning about the final positions as the search unfolds. We also point out that we decided to implement a *restaring mechanism* in SABIO in order to execute each instance from different restart points (i.e, assign different value selection strategies to unconstrained teams) because we identified in the tests for goal 2 that most of the instances were solved under a time limit of 0.51 seconds. This restart mechanism helped the search algorithm to scape from branches in the search tree where no solutions are found for certain time, thus having the effect of a kick move in order to explore other branches of the tree during search.

In order to train the classifier, we managed to define an array of 6 features (starting position, wanted position, direction and distance, fixtures to play, range rate, best executing strategy) to train a model that helps SABIO cover a wider range of problems on the set of instances that we created, however, we point out that the classifier that we trained was based on an 18 team competition and its behavior on smaller or bigger competitions might not be accurate because of the nature of the features. Interestingly, the new modifications presented an improvement from 64.35% to 94.3% solved instances with position in ranking constraints and the differences on average time of the former version of SABIO and the new version are not significant. The improvements also let the proposed model to reduced the amount of unsolved instances from 14 to 1 on final score constraints, despite the fact, we didn't focus our research in this kind of constraints given the low percentage of unsolved instances.

Finally, during this research we found out that previous soccer problems were

tackled using integer programming, therefore, we decided to present a MIP (mixed integer programming) version of SABIO in chapter 6. We use CPLEX to implement it and compare the performance of the new CP model and the MIP model. Additionally, we observed that the MIP and the CP approaches are complementary. Broadly speaking, the MIP model is able to solve more instances than the CP one, i.e., 1082 more than the pure CP approach and 171 more than the CP with machine learning. Alternatively, the CP approach is faster than the MIP approach. We decided then to propose a sequential execution of both models to improve SABIO's efficiency and effectiveness by exploiting the strengths of both alternatives. The mixed solver turned out to be about 72% faster than the pure MIP one, and 68% faster than the pure CP one.

In this document we have combined the use of constraint programming, machine learning, and mixed integer programming to propose models to solve general soccer fan queries at different stages of a competition. We presented four alternative solutions for the problem: CP, CP with machine learning, MIP, and a mixed approach between the MIP solution and CP with machine learning. Unlike previous problem dependent linear programming models, our approach also opens the possibility to represent different tournament and league formats (e.g., using single or double round robin game scheduling). Our computational experiments showed that our models can be solved fast and the use of mixed techniques such as MIP, CP, and machine learning lead to a very fast and robust solver. The improvements and the models presented here will have an impact on SABIO's productivity and competitiveness because it will improve its service quality and efficiency by covering a wider range of problems, thus bring a better service to its users who will take more advantage of the strengths that SABIO might offer.

7.2. Future Work

We finish this document by pointing out some further research opportunities that we identified during our research work:

- SABIO offers an interactive platform for user-defined queries to tackle soccer computational problems. This kind of platform can also be extended to other competitions based on points such as baseball, basketball, and handball.
- In this research, we implemented a mixed integer programming version of SABIO using CPLEX (V12.6.2) as our reference solver. It would be very convenient to implement versions of SABIO in different MIP solvers such as GLPK, LP Solve, CBC, etc. This solvers can be used to generate a benchmark and analyse the results in order to to select a free solver to implement the MIP version of SABIO and thus offer the service online.

- The mixed integer programming version of SABIO offers the opportunity to tackle soccer queries as optimization problems. With the MIP implementation it's possible to either maximize or minimize certain variables in order to satisfy user-defined constraints (e.g., maximum number of games that certain team can afford to lose and still qualify to the playoffs or minimum number of points that certain team has to get to have a chance to qualify to the playoffs). This kind of queries are natural in MIP, however, it would be interesting to implement the optimization version of SABIO using branch and bound in order to offer a wider number of queries and perform a series of tests to compare performance against the MIP version.

- A central question in comparing heuristic algorithms is whether one algorithm outperforms another because it is fundamentally superior, or because its developers more successfully optimized its parameters. Therefore, an open optimization step can be to perform a parameter tuning to both models CP and MIP using tools like ParamILS. Particularly, it would be interesting to study parameter tuning for module that SABIO has to configure the search (i.e., variable order and value selection heuristics).

- There are queries that users commonly ask about their teams. However, some queries might require certain expertise from the user in order to define them in form of constraints. It would be interesting to ease this process and add certain functionality to the web application with some default questions related to the elimination problem, guaranteed qualification problem, possible qualification problem, etc.

- Previous approaches like FUTMAX [59][61] used a framework to retrieve and publish information related to qualification and elimination in sports tournament over the Internet. SABIO requires to constantly update its databases in order to keep track of the soccer tournaments and leagues. However, this process is carried out manually and it could cover more competitions if the process is automated.

Bibliography

[1] F. Rossi, P. Van Beek, and T. Walsh, *Handbook of constraint programming*. Elsevier, 2006.

[2] M. Wallace, «Practical applications of constraint programming», *Constraints*, vol. 1, no. 1-2, pp. 139–168, 1996.

[3] K. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[4] K. Leyton Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, «A portfolio approach to algorithm selection», in *IJCAI*, vol. 1543, 2003, p. 2003.

[5] A. Arbelaez, «Learning during search», PhD thesis, Université Paris, 2011.

[6] L. Michel and P. V. Hentenryck, «A constraint-based architecture for local search», in *ACM SIGPLAN Notices*, ACM, vol. 37, 2002, pp. 83–100.

[7] J.-C. Régin, «Solving problems with cp: Four common pitfalls to avoid», in *Principles and Practice of Constraint Programming–CP 2011*, Springer, 2011, pp. 3–11.

[8] T. Bernholt, A. Gälich, T. Hofmeister, and N. Schmitt, «Football elimination is hard to decide under the 3-point-rule», in *MFCS*, 1999, pp. 410–418.

[9] W. Kern and D. Paulusma, «The new fifa rules are hard: Complexity aspects of sports competitions», *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 317–323, 2001.

[10] J. E. Borrett, E. P. Tsang, and N. R. Walsh, «Adaptive constraint satisfaction: The quickest first principle», in *Proceedings of the 12th ECAI*, Citeseer, 1996.

[11] S. A. M. Elsayed and L. Michel, «Synthesis of search algorithms from high-level cp models», in *Principles and Practice of Constraint Programming–CP 2011*, Springer, 2011, pp. 256–270.

[12] M. Milano and M. Wallace, «Integrating operations research in constraint programming», *Annals of Operations Research*, vol. 175, no. 1, pp. 37–76, 2010.

[13] P. Van Roy and S. Haridi, *Concepts, techniques and models of computer programming*. 2003.

[14] D. Mehta, B. O'Sullivan, L. Kotthoff, and Y. Malitsky, «Lazy branching for constraint satisfaction», in *ICTAI*, Nov. 2013.

[15] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen, «A survey of very large-scale neighborhood search techniques», *Discrete Applied Mathematics*, vol. 123, no. 1, pp. 75–102, 2002.

[16] W. D. Harvey and M. L. Ginsberg, «Limited discrepancy search», *IJCAI (1)*, pp. 607–615, 1995.

[17] M. Dorigo, V. Maniezzo, and A. Colorni, «Ant system: Optimization by a colony of cooperating agents», *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 26, no. 1, pp. 29–41, 1996.

[18] F. Glover, «Tabu search - part i», *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[19] ——, «Tabu search - part ii», *ORSA Journal on computing*, vol. 2, no. 1, pp. 4–32, 1990.

[20] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *et al.*, «Optimization by simulated annealing», *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[21] V. Černỳ, «Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm», *Journal of optimization theory and applications*, vol. 45, no. 1, pp. 41–51, 1985.

[22] P. Van Roy, D. Duchier, L. Kornstaedt, M. Homik, C. Schulte, and T. Müller, *Finite domain constraints*. [Online]. Available: `https://mozart.github.io/mozart-v1/doc-1.4.0/system/node26.html`.

[23] R. Dechter and I. Meiri, *Experimental evaluation of preprocessing techniques in constraint satisfaction problems*. UCLA, Computer Science Department, 1989, pp. 271–277.

[24] R. M. Haralick and G. L. Elliott, «Increasing tree search efficiency for constraint satisfaction problems», *Artificial intelligence*, vol. 14, no. 3, pp. 263–313, 1980.

[25] C. Bessiere and J.-C. Régin, «Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems», in *Principles and Practice of Constraint Programming—CP96*, Springer, 1996, pp. 61–75.

[26] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, «Boosting systematic search by weighting constraints», in *ECAI*, vol. 16, 2004, p. 146.

[27] I. P. Gent, C. Jefferson, L. Kotthoff, I. Miguel, N. C. Moore, P. Nightingale, and K. E. Petrie, «Learning when to use lazy learning in constraint solving.», in *ECAI*, 2010, pp. 873–878.

[28] T. M. Mitchell, *Machine Learning*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1997, ISBN: 0070428077, 9780070428072.

[29] A. Smola and S. Vishwanathan, *Introduction to machine learning*. 2008.

[30] E. Alpaydin, *Introduction to machine learning*. MIT press, 2014.

[31] B. O'Sullivan, L. De Raedt, S. Nijssen, and P. Van Hentenryck, Report from Dagstuhl Seminar 11201: Constraint Programming meets Machine Learning and Data Mining, 2011.

[32] L. Kotthoff, «Algorithm selection for combinatorial search problems: A survey», *AI Magazine*, 2014. [Online]. Available: `http://4c.ucc.ie/~larsko/`.

[33] B. A. Huberman, R. M. Lukose, and T. Hogg, «An economics approach to hard computational problems», *Science*, vol. 275, no. 5296, pp. 51–54, 1997.

[34] J. R. Rice, «The algorithm selection problem», 1975.

[35] C. P. Gomes and B. Selman, «Algorithm portfolio design: Theory vs. practice», in *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*, Morgan Kaufmann Publishers Inc., 1997, pp. 190–197.

[36] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, and B. O'Sullivan, «Using case-based reasoning in an algorithm portfolio for constraint solving», in *Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.

[37] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, «Satzilla: Portfolio-based algorithm selection for sat.», *J. Artif. Intell. Res.(JAIR)*, vol. 32, pp. 565–606, 2008.

[38] L. Pulina and A. Tacchella, «A multi-engine solver for quantified boolean formulas», in *Principles and Practice of Constraint Programming–CP 2007*, Springer, 2007, pp. 574–589.

[39] M. Birattari and J. Kacprzyk, *Tuning metaheuristics: A machine learning perspective*. Springer, 2009, vol. 197.

[40] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle, «F-race and iterated f-race: An overview», in *Experimental methods for the analysis of optimization algorithms*, Springer, 2010, pp. 311–336.

[41] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, «Paramils: An automatic algorithm configuration framework», *Journal of Artificial Intelligence Research*, vol. 36, no. 1, pp. 267–306, 2009.

[42] S. Minton, «Automatically configuring constraint satisfaction programs: A case study», *Constraints*, vol. 1, no. 1-2, pp. 7–43, 1996.

[43] S. L. Epstein, E. C. Freuder, R. Wallace, A. Morozov, and B. Samuels, «The adaptive constraint engine», in *Principles and Practice of Constraint Programming-CP 2002*, Springer, 2002, pp. 525–540.

[44] J. R. Quinlan, «Simplifying decision trees», *International journal of man-machine studies*, vol. 27, no. 3, pp. 221–234, 1987.

[45] R. Battiti and M. Brunato, «The lion way», *Machine Learning Plus Intelligent Optimization. Applied Simulation and Modelling*, 2013.

[46] P. R. Watkins, «Integer and combinatorial optimization», *Journal of the Operational Research Society*, vol. 41, no. 2, pp. 177–178, 1990.

[47] R. Bosch and M. Trick, «Integer programming», in *Search methodologies*, Springer, 2014, pp. 67–92.

[48] J. C. Smith and Z. C. Taskin, «A tutorial guide to mixed-integer programming models and solution techniques», *Optimization in Medicine and Biology*, pp. 521–548, 2008.

[49] S. Szymanski, *It's football not soccer*, URL: `http://ns.umich.edu/Releases/2014/June14/Its-football-not-soccer.pdf`, University of Michigan, 2014. [Online]. Available: `http://ns.umich.edu/Releases/2014/June14/Its-football-not-soccer.pdf`.

[50] W. Kern and D. Paulusma, «The computational complexity of the elimination problem in generalized sports competitions», *Discrete Optimization*, vol. 1, no. 2, pp. 205–214, 2004.

[51] B. L. Schwartz, «Possible winners in partially completed tournaments», *SIAM Review*, vol. 8, no. 3, pp. 302–308, 1966.

[52] A. Hoffman and T Rivlin, «When is a team "mathematically" eliminated?», Princeton, NJ: Princeton Symposium on Mathematical Programming, 1967, pp. 391–401.

[53] S. T. McCormick, «Fast algorithms for parametric scheduling come from extensions to parametric maximum flow», *Operations Research*, vol. 47, no. 5, pp. 744–756, 1999.

[54] I. Adler, A. L. Erera, D. S. Hochbaum, and E. V. Olinick, «Baseball, optimization, and the world wide web», *Interfaces*, vol. 32, no. 2, pp. 12–22, 2002.

[55] K. D. Wayne, «A new property and a faster algorithm for baseball elimination», *SIAM Journal on Discrete Mathematics*, vol. 14, no. 2, pp. 223–229, 2001.

[56] J. C. Guedes and F. S. Machado, «Changing rewards in contests: Has the three-point rule brought more offense to soccer?», *Empirical Economics*, vol. 27, no. 4, pp. 607–630, 2002.

[57] D. Gusfield and C. Martel, «The structure and complexity of sports elimination numbers», *Algorithmica*, vol. 32, no. 1, pp. 73–86, 2002.

[58] J. Christensen, A. N. Knudsen, and K. S. Larsen, «Soccer is harder than football», *International Journal of Foundations of Computer Science*, vol. 26, no. 04, pp. 477–486, 2015.

[59] C. C. Ribeiro and S. Urrutia, «An application of integer programming to playoff elimination in football championships», *International Transactions in Operational Research*, vol. 12, no. 4, pp. 375–386, 2005.

[60] D. Pálvölgyi, «Deciding soccer scores and partial orientations of graphs», *Acta Univ. Sapientiae, Math*, vol. 1, no. 1, pp. 35–42, 2009.

[61] C. Lucena, T. Noronha, S Urrutia, and C. Ribeiro, «A multi-agent framework to retrieve and publish information on qualification and elimination data in sports tournaments», *Rio de Janeiro, Brazil: Monografias em Ciência da Computação*, 2006.

[62] L. F. Vargas Rojas, «Diseño e implementación de una aplicación web y móvil para consultar sabio», Tesis de Pre-grado, 2014.

[63] F. Hutter, H. H. Hoos, and T. Stützle, «Automatic algorithm configuration based on local search», in *AAAI*, vol. 7, 2007, pp. 1152–1157.

[64] U. of Waikato. (2008). Attribute-relation file format (arff). Accessed: 2015-12-05, [Online]. Available: `http://www.cs.waikato.ac.nz/ml/weka/arff.html`.

[65] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, Third. Morgan Kaufmann, 2005, pp. 410–427.