École polytechnique de Louvain

General Game Playing and Constraint Programming to Model, Play and Solve any Logic Puzzles

Author : **Tom Doumont**
Supervisors : **Eric Piette**
Readers : **Damien Sprockeels**, **Peter Van Roy**, **Achille Morenville**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Contents

Acknowledgments

I would like to thank all the people who supported me in the completion of the present Master's thesis. First, I want to express my immense gratitude to my thesis supervisor, Eric Piette, for the quality of his advice and the infinite patience he has demonstrated. I am also grateful to Pierre Schaus, Achille Morenville and Damien Sprockeels for accepting to be part of this thesis jury.
My thank also go to my family, friends and my girlfriend for their support and for letting me talk about my thesis with them.
Finally, I would like to thank Noah, Ana and Téo for understanding that some days I just didn't have the time to play with them, even though I was at home.

ChatGPT was used sparingly when I struggled to produce clear sentences and should not be held responsible for any remaining errors.

1 Introduction

In spite of the differences in culture, language, and age, there is probably a sight familiar to all, at least in Asia, Europe, and North America. Surprisingly, this sight is of a mathematical object, a discipline that many claim to hate. It's a simple game played alone on a 9 by 9 grid, which can be drawn in a matter of seconds on a piece of paper. Though seemingly useless, it has experienced one of the greatest expansions since the invention of Chess thousands of years ago. By now, the astute reader should have easily guessed what I'm talking about: The Sudoku.

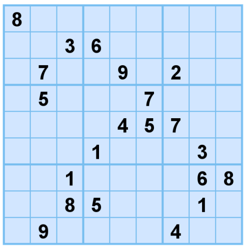
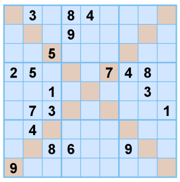
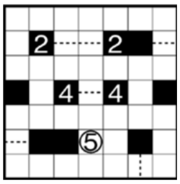


Figure 1.1: a Sudoku grid


Sudoku X


Suraromu

It is simple enough that the rules can be explained to a kid, and that most of us can remember them perfectly. Yet it is difficult enough that we have not proven some simple facts about it.[1] In fact, Sudoku is NP-complete when generalized to a n by n grid [2].

There are hundreds upon hundreds of similar problems [3][4][5]. Ranging from the simpler ones like Sudoku X to more esoteric ones like the Suraromu. In this work we will call them *Logic Puzzle*, and we will restraint ourself to perfect information, single-player games without planification (i.e: the moves to solve the puzzles can be played in whatever order, as opposed for example to the *Tower of Hanoi*[2] game).

[1]For example we don't know yet the maximum number of clues for a minimal Sudoku (i.e. a Sudoku such that removing any of the clues dissipates the uniqueness of the solution) [1].

[2]The tower of Hanoi is a game involving three pegs and a set of disk of different size. You have to move all the disk to a different peg, but you can only move one disk at a time and a disk can never be placed on top of a smaller disk [6].

Whereas creating a program to solve one specific logic puzzle such as Sudoku may be quite straightforward, it may not be a good idea to solve all of them in that fashion. Here are a few reasons why:

- It would be a time consuming process. There exist hundreds of logic puzzles. Writing a new program each time we want to solve one of them would be wildly inefficient. Many such puzzles incorporates similar elements and rules [7] and solving all of them from scratch would require to implement multiple times the same type of resolution process.
- Even if we solved all those problems, it would be in vain since it's relatively easy to create new similar games by inventing new rules from scratch. For example the YouTube channel "Cracking the cryptic" regularly introduces new game mechanics that are not present in the corpus we worked on [8] [9]. Moreover the existing rules can be easily combined to form new games. For example we can combine the rules of the Nonogram with a new rule ensuring that some regions contains a specific number of black cells (similar to what can be found in the *Killer Sudoku*) to create a completely new puzzle.
- It would be of a lesser interest for game designers, which need to be able to test a game without learning to code. Traditional actors of logic games, such as Nikoli[3], don't use computer generation because the problems created don't have the same "feeling" and are less enjoyable because they don't take into account the solver's possible reasoning. This results in problems where the difficulty is incorrectly distributed, with one or a few very hard first steps, and then a simple solution [10].
  However, in the case of Nikoli, most of their content is proposed by their reader [11]. It would be useful for this audience to validate their design without having to learn a programming language to model and solve it, regardless of whether they create a problem for an existing game or invent an entirely new game.

To try and offer a better solution to this type of problems, the present work proposes to use the *Ludii* general game playing system [12] to create a general game playing agent capable of designing and solving (almost) all logic puzzles. A general gaming system is a system allowing to model and play many different games, in this case via a ludemic human-like [13] representation of its rule. The aim of this agent is to use the techniques of constraint programming [14], a paradigm well suited for solving combinatorial problems, by transforming this human-like representation to an XCSP [15] instance. Then, to use a constraint programming solver to produce a solution to the problem and input it in Ludii.

In this thesis we will delve more in the details of this agent, and the choices that were made during its conception, such as the solver utilized on the instance or the internal implementation of the constraints. Moreover we will evaluate the performances of our solver, both in term of number of solved problems and in term of resolution speed (Depending on the size of the instance). The main issues we faced were that some constraints used

[3]Nikoli is a Japanese publisher who specialize in logic puzzle that became prominent worldwide with the popularity of the Sudoku, their site can be found at: https://www.nikoli.co.jp/en/

frequently in the puzzles don't translate easily to constraint in constraint programming. This was challenging for pattern-based games like Nonograms, but we managed to solve them. However, we were unable to solve games that involve graph theory, especially those with connectivity constraints, such as SlitherLink.

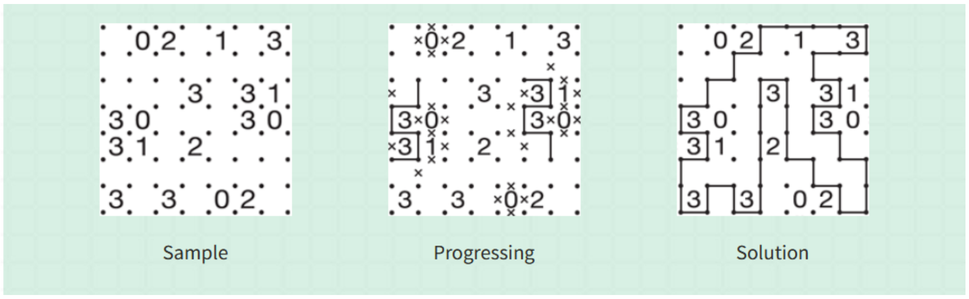


Figure 1.2: SlitherLink, a game in which hints indicate the number of neighboring edges that have to be colored, but all colored edges have to be connected.

The following chapters are organized as follows. Chapter 2 discuss in greater depth the Ludii framework, constraint programming, and the foundational works upon which this thesis is based. Chapter 3 details the implementation process and explains the selection of specific constraint programming solvers. Chapter 4 presents the performance results of the agent in terms of number of problems solved, speed and generalization. Finally, Chapters 5 and 6 offer perspectives for future work and present the main conclusions of this thesis.

2 Background and previous works

In this section we will describe the tools used in this thesis, the Ludii framework, the constraint programming paradigm and the XCSP formalism. Moreover, we will briefly analyze the previous works realized on this subject.

2.1 General Game Playing and Ludii

2.1.1 General Game Playing

Most of the early research on artificial intelligence focused on specific games such as Chess or Drought. However, this endeavor has limited value in AI [16]. General Game Playing (GGP) is the study and construction of agents which are not limited to a single game, but can play effectively any game given a correct representation of their rules. A General Game Playing system is the framework or environment designed for the creation and execution of general game players (AI agents). There are many such systems and it's important to carefully study them in order to select the appropriate one for our use case.

2.1.2 Game Description Language

One of the tools that allowed the recent evolutions of GGP was the introduction of GDL [17] (Game Description Language), a programming language created for the express purpose of describing rules of games. It was designed by a team at Stanford University in 2006 in order to cover finite, discrete, deterministic multi-player games of complete information. It uses the syntax and logic of first-order logic programming, it's built upon Datalog, a variant of Prolog, which is a logic programming language that uses facts, rules, and queries.
Since then it has been one of the main focus in the research on General Game Playing. Notably by being the language used by the International General Game Playing Competition [18] hosted on GameMaster [19] until 2016. Interestingly, the last International General Game Playing competition was won by WoodStock [20], which is a solver based on stochastic constraint programming, as described in [21].

Later, GDL has also been extended to include imperfect-information games [22] and epistemic games [23]. Finally the use of Propositional Networks [24] has allowed the speed of the reasoning process to improve by several orders of magnitude compared to custom-made or Prolog-based GDL reasoners.
However GDL has some severe limitation [12]. Key aspects of games must be defined from scratch every time we want to use them. Implementing new games is time consuming and require a solid background on first order logic and computer sciences, as a consequence the library of games available in GDL has only grown with a handful of new games every year. The description doesn't encapsulate key game elements that human think about when describing games, and existing games need a lot of change to apply a small change of rules (for example when changing the size of the board). The processing speed remains a limiting factor of many applications as it requires logic resolution.

2.1.3 RBG

More recently RGB was developed as an alternative to GDL to allow the modelisation of more complex games with large branching factor [25]. It aims to compensate certain drawbacks of pre-existing languages. It is based on an idea developed in [26] to represent a subset of games with regular languages (i.e: A regular language being a language that can be recognized by an automaton). However as the expressions utilized are simplistic and only applied to one piece at a time, this representation are limited and cannot represent non standard comportment. RBG expands on this idea to describe the full range of deterministic board games.
RBG is composed of two languages, one low level to allow the efficient computation by the compiler and one high level to allow a higher readability. This allows it to mostly outperform other general game playing framework [27] in term of efficiency. The way this is evaluated is by measuring the number of flat Monte Carlo playouts by seconds. (i.e: The number of games per second we can finish while playing randomly). It can describe every finite deterministic game with perfect information, but not game with incomplete information, which is thankfully not a problem when considering deduction puzzles.

2.1.4 Ludii

Ludii is a general game playing system aiming to model, play and design a large variety of games, such as board games or logic puzzles [12]. It contains more than a thousand different playable games. Current efforts focus on finding an efficient GGP model for games with imperfect information, such as card games [28]. Furthemore the Ludii language has been shown to be universal [29], it can represent any table game, including non-deterministic and imperfect-information ones.
It is being developed as part of the Digital Ludeme Project [30] which aims to improve our understanding of traditional games using modern AI techniques, to study their historical development and the spread of ludic ideas. Good examples of such an utilization of artificial intelligence to study historic games can be found in [31] and in [32] where it is used to evaluate key metrics about a specific set of rules in order to shed light on potential rules differences or properties of the games.
The Digital Ludeme Project is carried out by the GameTable COST Action network, an

international network of scholars to inspire methodologies and applications on how to use game AI to study, reconstruct, and preserve the cultural heritage of games [33][34].

It utilizes an approach based on ludeme [35], a term widely used in the research community surrounding games including AI, to describe an unit of game-related information. Characteristics of ludemes as defined by [35] are:

- They are discrete: They are understandable as a single discrete unit of information, even if they can encapsulate other concept. (For example the move of a knight is a single ludeme, even if it encapsulates other concepts).

- They are transferable: They can be transferred from one game to another without loosing their meaning.

- They are contrastive: Changing a ludeme in a game should change the game itself, for example on a square grid we can define the move of a knight either as "An “L” walk (forwards, forwards, left/right, forwards) in any of the four orthogonal directions" or "Closest non-adjacent cell of a different color". Since these two definitions produce strictly the same result, they are not two distinct ludemes.

- They are compound: A ludeme can be composed of multiple contrastive sub-elements that are also ludemes. It is obvious when comparing rules like "Hop over any adjacent piece" and "Hop over any adjacent piece to flip it"

It was with this definition of a ludeme in mind that was developed the *Ludemic model of games*, a computational method for describing games by their elements. It was created as part of the Ludi software system then improved for Ludii [12]. In this model, game are composed of three types of elements:

- Class name: Denoted by bracketed lowercase keywords (players _), they corresponds to a Java Class in Ludii’s code base.

- Attributes: Denoted by uppercase letters (Line), they describe a single constant value.

- Variables: Denoted by numbers, strings or booleans ("Disc" or 3), they represent a single user-set value.

This model is based on a grammar automatically generated from the Java class in the Ludii code base using a *class grammar approach* [36]. A *class grammar* as introduced in [36] is a formal grammar derived directly from the class hierarchy of the underlying source code, java in this case. It aims to facilitate game design by providing a simple interface to the full functionality while hiding the implementation details.

Formally, the grammar is an Extended Backus-Naur Form style grammar consisting of a set of production rules, each corresponding to exactly one java class. The full grammar can be found in [37] and details on how this grammar is used to generate game representations and the model necessary to play them can be found in [13]. As an example to study the ludemic model and Ludii’s syntax we can examine the implementation of Tic-Tac-Toe in Ludii:

```
(game "Tic-Tac-Toe"
    (players 2)
    (equipment { (board (square 3)) (piece "Disc" P1) (piece "Cross" P2) })
    (rules
        (play (move Add (to (sites Empty))))
        (end (if (is Line 3) (result Mover Win)))
    )
)
```

We can see that a Ludii game is composed of a 3-tuple of ludemes: *<Players, equipment, Rules>*:

- **Players:** This element defines the participants and the game's control flow. In this case players play Alternatively, but they could play simultaneously for games like Rock-Paper-Scissors. For all logical puzzles as described in [7] there is always only one player.
- **Equipment:** This includes a list of containers and a list of components. Containers represent the game's layout as graphs, where playable sites are vertices, faces and/or edges. They define the geometry of the board. In the previous example, the board is a 3 by 3 square, but any board that can described as a combination of vertices, edges and/or cells can be defined [38].
  Components are the pieces, cards, tiles or dice that can be placed on the sites of the containers. In this case the components are the pieces "Disc" and "Cross" which are not present at the start of the game.
- **Rules:** The rules consist of a 3-tuple *<Start, Play, End>*:
  - Start denotes a list of actions that executes before the start of the game. For logic puzzles it is mainly used to apply a different *Challenge* to the game, i.e., the initial size of the board and position of the hints specific to this instance of the puzzle.
  - Play allow to calculate the set of all possible actions given a particular state. We only want the player to be able to play moves that are allowed by the rules of the game. To do so we generate a list of possible moves and check that he chooses one of them, in the aforementioned game a player can drop a piece on any empty site. It is in this ludeme that we will define the different constraint that the puzzle will have to respect.
  - End denotes a set of condition under which a given state is terminal, each condition leading to a score vector to evaluate the outcome of the game. For the Tic-Tac-Toe it is straightforward and simply state that if a line of 3 pieces is formed, the current player wins. Most of the time the end condition for puzzles games will be that every variable is attributed and every constraint is satisfied, which is conveniently described by the ludeme *(is Solved)*. The outcome of the game is then invariably that the only player wins.

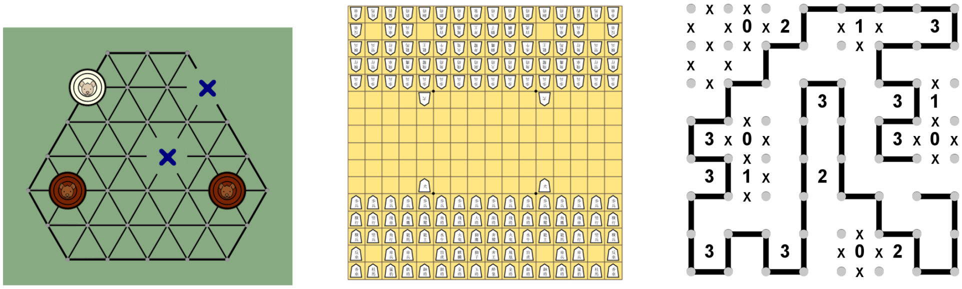


Figure 2.1: Some examples of games in Ludii

Ludii has some clear advantages over other General Game Playing systems [39][12]:

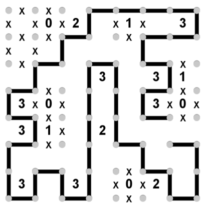
- Clarity/Simplicity: Code is self-explanatory even for non-programmer, easier to create and modify.
- Generality: Ludii can theoretically support any game that can be programmed in java (which is any game). While GDL and RBG are restricted to deterministic perfect information games. This can be a severe limitation as it excludes most card games and games necessitating dice.
- Extensibility: The structure of Ludii make it very easy to add new ludemes as we just have to add a class to the ludeme library.
- Evolvability: GDL uses complex chains of logical operations that don't allow the use of genetics algorithms to create new games. With Ludii, this can be done and has been done [40].
- Cultural applications: Ludii is linked to a server and database that stores relevant cultural and archaeological information about the game [41].
- Efficiency: Efficiency is not a core aspect of Ludii, we sometimes prefer to favor clarity and evolvability. Whatsoever, Ludii still outperforms RGB interpreter and GDL. The RBG compiler still outperforms Ludii but most of the time their performances are of the same order of magnitude.
- Existence of previous works: A preliminary work [42] already exists that implements some core games and concepts of logic puzzles in the Ludii framework.

From this comparison of the most commonly used General Game Playing languages and frameworks, Ludii emerges as the most suitable for our application. While it lacks the extensive literature of GDL and may not always match the speed of RBG, these shortcomings do not appear to be critical for our purposes. Moreover, Ludii offers clear advantages in terms of extensibility and simplicity, which will be essential for implementing the new features [43].

2.1.5 Model and play logic puzzles within Ludii

In this master thesis [44], Pierre Accou was tasked to extend the library of logic problems in Ludii. To do so he uses [7] to define rigorously what is a *deduction puzzle*. It is worth noting that some of the terms used in this paper are also utilized as general game concept in Ludii [45]. He then proceeds to distinguish deduction puzzles and planning puzzles, the latter requiring a strategy to anticipate the moves needed to solve the puzzles, the order of the movements being a key part of the solution. In a deduction puzzle, the movements can be played in any order. Deduction puzzle can be further divided in three categories:

- Path Puzzles: Puzzles that require a path, connections or arrows between elements.


(a) SlitherLink

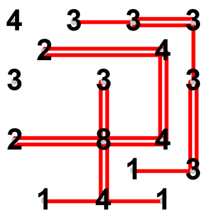
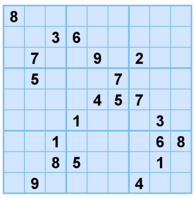

(b) Hashiwokakero

Figure 2.2: Path puzzles

- Position Puzzles: Puzzles that requires some numbers, letters or symbols to be placed at specific positions.


(a) Sudoku

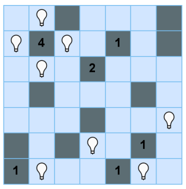
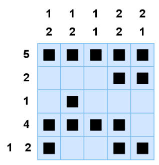
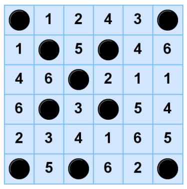

(b) Akari

Figure 2.3: Figure puzzles

- Shading Puzzles: Puzzle where the player has to color certain elements to find the solution.



(a) Nonogram



(b) Hitori

Figure 2.4: Shading puzzles

In this thesis, the author attempts to implement all the games from Nikoli[1] and Cross+A[2] to expand Ludii's catalog, aiming to cover as many puzzle types as possible. In doing so, a comprehensive test set is incidentally created, which can be used for the development of a general game-playing agent for these types of puzzles.

Some of these games were already implemented in Ludii, while others were easily added using the existing code base. However, the addition of new ludemes and modifications to existing ones were necessary to accommodate the new games. The following sections present a selection of both old and new games relevant to this work, along with the old and new ludemes used to describe them.

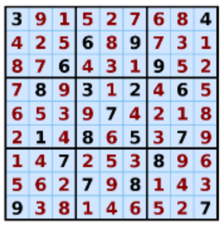
2.1.6 Games of note

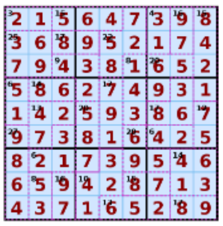
As of the writing of this thesis, the version of Ludii produced by Pierre Accou incorporates more than 40 different logic puzzle. Describing them all here would be of little interest. Similarly their rules can be complex, we will thus focus on game of particular interest on the context of this thesis and only develop the aspect of their rules that differ from other similar games.

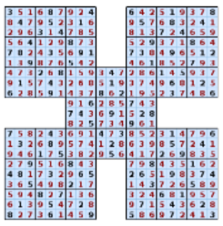
Sudoku and its variants

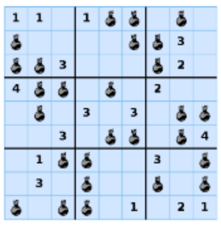
One of the main category of deductive puzzles in Ludii are variants of Sudoku. They have various shapes and sizes, but mostly they can be describe as a set of region on the board where each of the location has to contain a distinct value. You can see below a selection of variants of Sudoku:

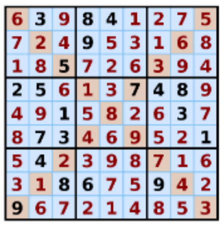
[1]https://www.nikoli.co.jp/en/puzzles/
[2]https://www.cross-plus-a.com/fr/index.htm

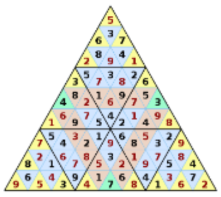

(a) Antiknight Sudoku

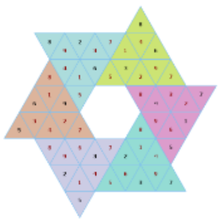

(b) Killer Sudoku


(c) Samourai Sudoku


(d) Sudoku Mine


(e) Sudoku X


(f) Tridoku


(g) Hoshi

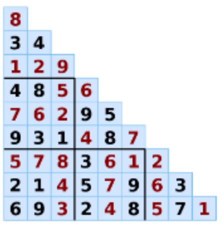

(h) Sujiken

Figure 2.5: Variants of Sudoku

Kakuro, Magic Square and assimilated

In Kakuro, there is a constraint on region that state the sum of the elements of a region must equal a specific value given as an hint at the start of the game. This rule is frequent in logic puzzle and is notably used in Magic Square and Killer Sudoku.

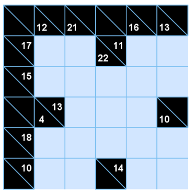


Figure 2.6: A Kakuro Grid

Nonogram and its variant

For games like Nonogram each row and column of the grid is associated with a set of numbers that indicate the lengths of consecutive blocks of filled cells in that row or column. For example, a clue of "3 1" for a row means there is a block of 3 filled cells followed by at least one empty cell, and then a block of 1 filled cell. Variant include different shapes of Nonogram (Like the Hexagonal Nonogram) and color Nonogram (Where clues have different colors).

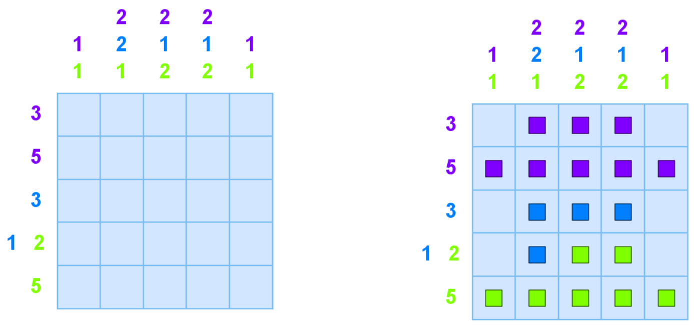
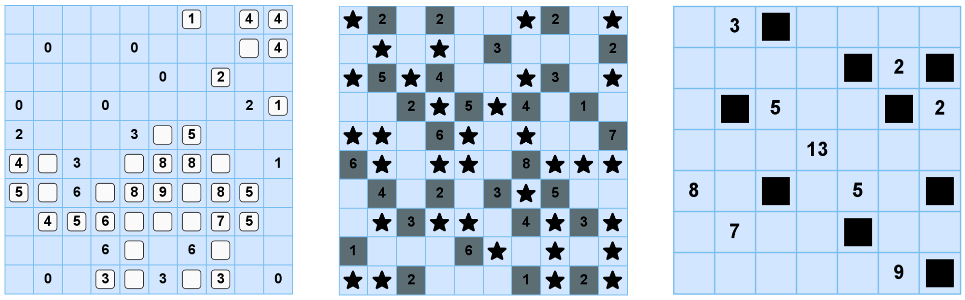


Figure 2.7: A color Nonogram and its solution

Problems counting the number of elements in a region

The main constraint in this large array of puzzles is that the number of a specific element in a specific region must be equal to a particular value. This can be straightforward, for example in Fill A Pix each number indicates the number of neighboring cells that are shaded. On the contrary it can be quite complex, for example in Buraitoraito, the hint in a cell indicates the number of stars in a region including all the cells orthogonal to the hint and bounded by other hints. Moreover sometimes the region is not statically defined, in Kurodoko for instance, the region used to evaluate the hint is bounded by the moves of the player.



(a) Fill-A-Pix (b) Buraitoraito (c) Kurodoko

SlitherLink, Big Tour, Masyu, Hashiwokakero

These puzzles are path-based problems, played on the edges of a graph. To solve them, the player must toggle specific edges to create a graph that meets certain conditions. Common constraints include requirements for the graph to form a single loop, to remain connected, for each cell to have a specific number of edges around it, and to prevent edges from crossing.

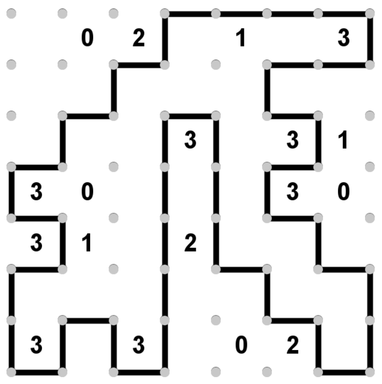


Figure 2.8: In SlitherLink the hint on cells indicates the number of toggled edges around it

2.1.7 Ludemes of note

In this section we will present the most frequently used ludemes in the implementation of puzzle present in Ludii. We will also present some ludemes that are only used in one specific puzzle, which sometimes happens when a rule is too specific to be covered by existing ludemes.

IsSum/AtMost/AtLeast

These three ludemes are applied on a region of the board. They compute the sum of the elements on this region, then they assure that this sum is equal/smaller/larger then another element that can be an integer or a function.

IsCount

This ludeme counts the number of specific element in a region, then it checks that it is equal to another element that can be an integer or a function.

IsMatch

This ludeme is primarly used to describe the Nonogram. It is applied on a list of cell and ensure that the values in these cells respect the pattern given in the ludeme. It is a direct implementation of the rule of the Nonogram described earlier.

IsConnex

This ludeme compute the number of components in a graph, and ensures that it doesn't exceed the authorized number.

IsCrossed

This ludeme is only utilised in the game Hashiwokakero, it checks that two vertices that cross each other can't be selected at the same time.

IsTilesComplete

This ludeme checks that in a region either all the cells are attributed, or none are.

IsDistinct
This ludeme is used with a region and an integer, it checks that this integer is unique on this region.

AllHintDifferent
This ludeme is only utilized in Hitori, which contains the unique rule that the player has to hide numbers written on the grid so that each number in columns and grids are different.

2.1.8 LudemPlex

LudemPlexes are shortcut for rules that are frequent in different games or puzzle in Ludii in order to facilitate the readability of the ludemic representation. In his Master's Thesis, Pierre Accou implemented a few of these LudemPlexes:

- **"OnlyOneWay":** This ludemplex used in graph problems checks that the puzzle is solved and that there is only one group of edges, i.e., the graph composed of these edges is connected.
- **"NumberOfEdgeByVertex":** In graph problems, this ludemplex ensures that the number of edges connected to a specific vertice is equal to 0 or 2. In practice, this imposes that the graph will form loops as imposed in Masyu, Slitherlink or BigTour.
- **"PieceNotAdjacent:"** This ludemplex ensure that two pieces cannot be placed adjacently in Hitori and Kurodoko.

2.2 Solving logical puzzles

2.2.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [46] is a heuristic search algorithm used for decision-making processes in game playing, it has been proven as very effective for playing games with very large branching factor such as Go, in fact MCTS was part of the algorithm used by the first artificial intelligence to beat the best human player [47]. It combines the principles of random sampling and tree search to find the most promising moves. It constructs a tree where each node represent a game state, select randomly a node from this tree, then simulate a lot of games starting from this node by playing randomly. It then update the statistics of the node for example with the number of victories/defeats. Once this process has been repeated enough times, the algorithm has a good statistical representation of the most promising moves and it can choose the most promising one. Some variation of this algorithm, such as UCT [48] offers informed way to choose the branches that further improve the efficiency the MCTS agents.
MCTS tends to perform well in General Game Playing because it does not require any game-specific knowledge such as heuristic evaluation function for the assessment of position, it simply needs to be able to simulate many games in short succession. Furthermore it can be parallelized with great efficiency. We can argue that Monte-Carlo Tree Search has been the most important technique introduced in GGP [49]. In the context of Ludii MCTS was one of the principal AI approaches that allowed the study of games in the Digital Ludeme Project [50][51].
Given these earlier success, it is natural to try to implement similar techniques to extend

them to single-player games. That's exactly what was done by [52] and [53]. They explore the use of Nested-Monte Carlo Tree Search in the resolution of logic puzzle. Nested-MCTS is a variant of MCTS allowing MCTS instances to run within each node of the main search tree. They show very robust results, particularly on planification problems. However they do not tend to perform very well en deduction puzzle, for example [52] performs badly when applied to the simple 9x9 Sudoku. [53] uses a constraint programming approach, but then uses MCTS as an heuristic for searching a solution (With the the number of variables that have been assigned before finding a variable with an empty domain before backtracking as a score). This result is really promising and we will compare it to the solution provided by more generic solvers.

2.2.2 Constraint Programming

Constraint programming is a paradigm of programming for solving combinatorial problems where the solution must satisfy a set of constraints or conditions [54] [14]. The general idea is that the user states the constraints and a general purpose solver is used to solve them.
A problem in Constraint Programming is composed of:

- Variables: The elements we have to determine to solve the problem, in this case typically representing a value the player has to choose for a specific location. For example in Sudoku, there would be a variable for each blank case at the start of the game.
- Domains of Value: For each variable, the range of values that it can takes. In the case of the classical Sudoku, it would be the integers from one to nine.
- Constraints: A set of relations between subsets of the variables, such as a relation specifying that each variable in a specific line of a Sudoku must be different.

Conceptually, when a solver is called in a Constraint Programming problem, it assigns a value to a variable from its domain, then *propagates* the constraints involving that variable. Propagation adjusts the domains of other variables connected by constraints, reducing possible values based on the change. If a variable's domain is altered due to propagation, the solver propagates the related constraints as well [14]. At the end of this process, three outcomes are possible:

- The problem is solved, so the solver stops and returns one of the solutions.
- A contradiction occurs (a variable's domain becomes empty, meaning no valid value exists), and the solver excludes the initial value assigned to the first variable, this is called Backtracking.
- No contradiction is encountered but the problem is not solved, so we repeat the process with another variable.

A Constraint Programming (CP) solver systematically explores the solution space. This guarantees that if a solution exists, it will eventually be found. Conversely, if no solution exists, the solver can rigorously prove it. Additionally, when a solution is identified, the method used to find it is fully traceable, offering high explainability.

Problems that are well-suited for Constraint Programming (CP) generally share several characteristics that make them ideal for this approach. They are combinatorial problems, with discrete domains and well defined constraints where finding any feasible solution is more important than an optimal one (Even though there are other paradigms for Constraint Optimization Problems). Logic puzzles have all those characteristics and are perfect candidates for Constraint Programming.
Moreover many such problems have been proven to be NP-complete [55], and Constraint Programming performs well on this class of problems [56][57].
All in all, it seems that Constraint Programming is a good tool for solving logic puzzle, and a great part of recent researches uses it when solving them [56] [57] [58] [42] .

2.2.3 XCSP

To allow for a common representation of constraint programming problems, a standard format was proposed in [59] and has been since extended to XCSP3 [15]. It employs the Extensible Markup Language (XML), which is a widely used flexible text format. It is commonly employed, and is a a fundamental element of HTML, which should make the XCSP3 format easy to understand for people who any person who has a background in information technology. You can see in the following text an example of the XCSP3 representation of a simple 4 by 4 Sudoku:

```xml
<instance format="XCSP3" type="CSP">
    <variables>
        <array id="x" note="x[i] is the cell i" size="[16]"> 1..4 </array>
    </variables>
    <constraints>
        <intension> eq(x[0],3) </intension>
        <intension> eq(x[2],2) </intension>
        <intension> eq(x[7],1) </intension>
        <intension> eq(x[10],1) </intension>
        <intension> eq(x[14],3) </intension>

        <allDifferent> x[0] x[4] x[8] x[12] </allDifferent>
        <allDifferent> x[1] x[5] x[9] x[13]</allDifferent>
        <allDifferent> x[2] x[6] x[10] x[14]</allDifferent>
        <allDifferent> x[3] x[7] x[11] x[15]</allDifferent>

        <allDifferent> x[0..3] </allDifferent>
        <allDifferent> x[4..7] </allDifferent>
        <allDifferent> x[8..11] </allDifferent>
        <allDifferent> x[12..15] </allDifferent>

        <allDifferent> x[0..1] x[4..5]  </allDifferent>
        <allDifferent> x[2..3] x[6..7] </allDifferent>
        <allDifferent> x[8..9] x[12..13] </allDifferent>
        <allDifferent> x[10..11] x[14..15] </allDifferent>
    </constraints>
</instance>
```

XCSP3 has many interesting properties [15], it is well documented and easy to understand, it is flexible and support a large range of constraints and frameworks. In fact, it should be able to model practically all constraints that can be found in major constraints solvers [60].

Furthermore a large ecosystem exists around XCSP3, with parsers written in Java and C++, as well as a python library and a java API for modeling constrained problem. Many series of instance are available and a competition of constraint programming is held annually using this format [61], competition to which most of the modern solvers participate, ensuring they remain XCSP3 compatible.
In order to compare performances of different solvers in the instances produced by our solver, we need a way to offer a common representation of the problems that the solvers can process. XCSP3 comes as a natural choice given the place of standard it occupies in the field of constraint programming.

2.2.4 Existing constraints in XCSP

There exist many constraints in XCSP, and we will focus here in the ones that are used by Ludii while creating XCSP instances.

Intension

The intensions constraints denotes a vast array of constraints that can be defined as Boolean expression, usually called Predicates. Predicates are represented under functional forms in XCSP3. The constraint x+y=z is written as eq(add(x,y), z). Many different operators can be used in an intension constraint, like arithmetic operators (addition, opposite, multiplication, remainder, minimum, ...), logic operator (Not, and, or, xor, ...) and relational operators (Less than, greater or equal, different from, ...). Intension constraints often represents the bread and butter of constraint programming and used to represent the most basic constraints. A simple intension constraint on the sum of two variables would be written like this:

```
<intension>
  <function> eq(add(x1,x2),x3) </function>
</intension>
```

Of course intension constraint can be arbitrary complex and express far more complicated constraints.

Extension

Extension constraints, also called table constraint, define explicitly the list of tuples allowed (Positive table constraints) or forbidden (Negative tables constraints) for a tuple variables. Conceptually, this is equivalent to enumerating the allowed/forbidden values for the tuples. For example the following code imposes that the tuple <y1, y2, y3> takes a value among the "supports".

```
<extension id="c2">
  <list> y1 y2 y3 </list>
  <supports> (0,1,0)(1,0,0)(1,1,0)(1,1,1) </supports>
</extension>
```

Counting

Counting constraints are based on summations or counting the number of times variables satisfy a certain conditions.
The summation constraint is used to verify that the sum of the values of variables meet a certain condition, such as being equal to or less than a specified value (That can be explicitly

stated, a variable itself or even a list of values, meaning that the sum of the variables must be equal to one element of this list). Moreover, a multiplicative coefficient can be applied to the different terms of the sum. This type of constraint would be written like this:

```
<sum>
  <list> y1 y2 y3 y4 </list>
  <coeffs> 4 2 3 1 </coeffs>
  <condition> (in,2..5) </condition>
</sum>
```

The count constraint ensures that the number of variables in a list that takes a specific values on a list respects a numerical condition. For example it can be utilized to imposes that the number 5 doesn't appears more than 9 times in a list of variables. We will see that this constraint is really useful while modeling logic puzzles.

AllDifferent

All different constraint are ubiquitous in logic puzzle and constraint programming in general. More then half the games implemented on Ludii use it one way or another. It applies on a list of variables and imposes that all take a different value. We can also add an exception, for example to allow multiple zeros in the list, which is useful for several puzzles. An example of this constraint could be:

```
<allDifferent>
   x1 x2 x3 x4 x5
  <except> 0 </except>
</allDifferent>
```

An other interesting variant is AllDifferent-list, which imposes that multiple list of variables vary by at least one element.

Regular

The constraint regular ensures that the sequence of values assigned to the variable form a word that can be recognized by a finite automaton[62]. We can examine one regular constraint in XCSP:

```
<regular>
  <list> x1 x2 x3 x4 x5 x6 x7 </list>
  <transitions>
    (a,0,a)(a,1,b)(b,1,c)(c,0,d)(d,0,d)(d,1,e)(e,0,e)
  </transitions>
  <start> a </start>
  <final> e </final>
</regular>
```

And the corresponding finite automaton:

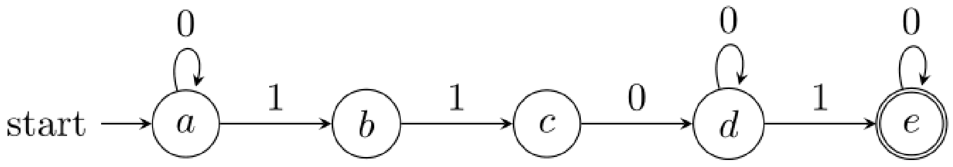
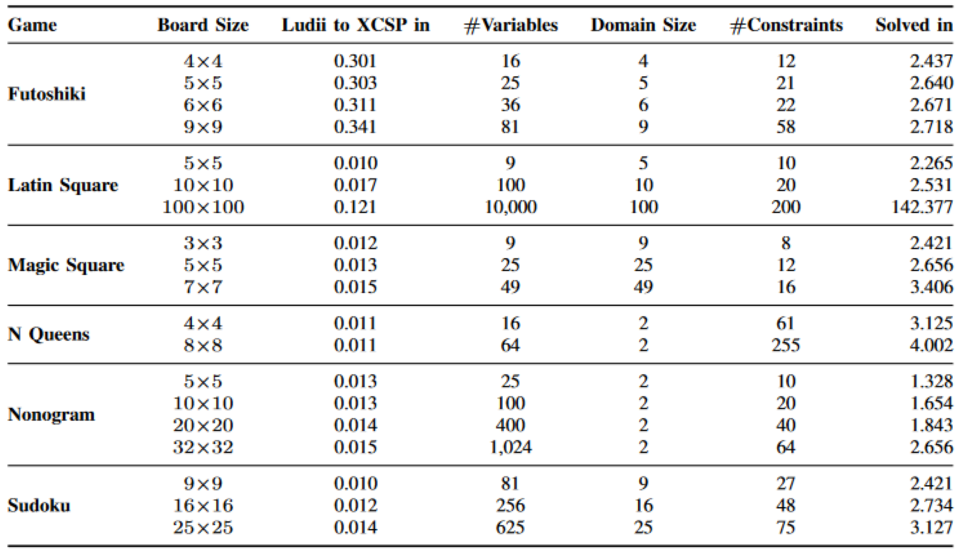


Figure 2.9: The automaton

We can see that the final automaton possess 5 states, ranging from a to e. The starting state is a and the final state is e. The key concept to understand this constraint is that variables are **transitions** in the finite automaton. The constraint is satisfied if the current state of the automaton after all transition is a final state. In the particular case of this automaton with the variables [x1 .. x7], the sequence (0, 0, 0, 1, 1, 0, 0) doesn't satisfy the constraint because if we follow those transitions the final state is d. Thus we know the variables [x1 .. x7] can't have the values (0, 0, 0, 1, 1, 0, 0). One valid attribution of the variables would be (0, 0, 0, 1, 1, 0, 0). The interest of this constraint for our use case is not immediately obvious, it's use will be discussed in more details in the implementation of games like the Nonogram.

2.2.5 Previous works

The present work relies heavily on an earlier paper titled "Ludii and XCSP: Playing and Solving Logic Puzzles" [42] written in 2019 by a team from Maastricht university. This paper details how it used the Ludii framework to produce an XCSP instance and solve it using a specific constraint programming solver called Abscon [63]. This solution was tested on six games as a prove of concept (Futoshiki, Latin Square, Magic Square, N Queens, Nonogram, Sudoku).
Then using this benchmark, the authors evaluated the performances of this generic solver on different sizes of board: As we can see, the translation from Ludii to XCSP is almost



Game	Board Size	Ludii to XCSP in	#Variables	Domain Size	#Constraints	Solved in
Futoshiki	4×4	0.301	16	4	12	2.437
	5×5	0.303	25	5	21	2.640
	6×6	0.311	36	6	22	2.671
	9×9	0.341	81	9	58	2.718
Latin Square	5×5	0.010	9	5	10	2.265
	10×10	0.017	100	10	20	2.531
	100×100	0.121	10,000	100	200	142.377
Magic Square	3×3	0.012	9	9	8	2.421
	5×5	0.013	25	25	12	2.656
	7×7	0.015	49	49	16	3.406
N Queens	4×4	0.011	16	2	61	3.125
	8×8	0.011	64	2	255	4.002
Nonogram	5×5	0.013	25	2	10	1.328
	10×10	0.013	100	2	20	1.654
	20×20	0.014	400	2	40	1.843
	32×32	0.015	1,024	2	64	2.656
Sudoku	9×9	0.010	81	9	27	2.421
	16×16	0.012	256	16	48	2.734
	25×25	0.014	625	25	75	3.127

Figure 2.10: Time necessary to generate and solve puzzle with Ludii, XCSP and Abscon

negligible, staying well under one second for very large instances. Even though the resolution time increases significantly for very large instances, all problems of sizes that are feasible for humans to solve are resolved within a few seconds. The authors do not offer a direct comparison, but we can reference the paper [58], which uses a specific algorithm called Deductive Search, a breadth-first, depth-limited, constraint-based approach.

We can observe that in the case of Sudoku, the only game common to both datasets, the resolution speed in the cited paper is around 0.1 seconds, whereas in Ludii, the resolution speed was closer to two seconds for a 9x9 Sudoku. This discrepancy can be explained by the hand-crafted constraint problem designed in the second paper. It is also worth noting that for one specific puzzle, which was not suited for the Deductive Search proposed in the paper, the solver reached a resolution time of over 10 seconds.

However, the difference in resolution time is not alarming for a general game-playing agent, since it does not scale much with the size of the problem, allowing the solver to be used efficiently for almost all practical purposes. For instance, a Futoshiki puzzle with 81 variables takes only slightly longer to solve than one with 16 variables (2.718 seconds and 2.437 seconds, respectively).

Towards the end of the paper, the team proposes extending Ludii's library of logic puzzles to the full range of Nikoli games. This was partially realized in [44], but the solution implemented at the time of the paper was not functioning in the current version of Ludii. The authors also suggest that it would be interesting to explore deductive search algorithms, such as the one presented in [58], to further improve performance. Moreover, they develop the idea that it would be useful for Ludii users to request hints. The combination of these two points is particularly interesting because the deductive search algorithm aims to emulate human processing power and the methods by which humans solve problems.

3 Implementation

3.1 Choice of the solvers

We had to select a subset of solvers in order to compare their efficiency in our puzzles. To select those solvers we used a few main criterion.
Firstly, we can study the performance of each solver by examining the result of the XCSP competitions [61][1]. XCSP competitions are organized each year since 2017 to evaluate the performances of Constraint Programming solvers. They evaluate the efficiency of solvers on six different categories, but two are of particular importance to us, the CSP competition and the miniCSP competition, who evaluates performances of solver on problems who only used a small subset of constraint.
Secondly, we want the solver to be able to process XCSP problems, thankfully all solvers participating in the XCSP competition have this ability.
Thirdly, we want the solver to integrate correctly with the Ludii player, in order to keep its utilization simple for users and limit the size of the executable.

3.1.1 ACE

ACE [64] is an open-source constraint solver developed by Christophe Lecoutre from the University of Artois in Java. It has constantly produced really good results in XCSP competitions, even though it is officially classed as "off-competition" since at least 2020 as C. Lecoutre is the person that select instances for the competition.
Furthermore ACE provides a *.jar* file to solve XCSP instance, which greatly facilitates the integration of the solver in Ludii.

3.1.2 Choco

Choco [65] is a Java library for constraint programming which was created in the early 2000s. It performed decently well in the recent editions of the XCSP competition. It aims to provide a complete and

[1]https://www.xcsp.org/competitions/

stable solver for commercial and academic usages. It incorporates a large array of different search strategies and constraints (up to 100). More importantly, it offers the possibility to easily define new search strategies. This could be interesting for further researches in the efficiency of different explorations algorithms in the case of puzzle problems.

3.1.3 AbsCon

The AbsCon solver [63] is the predecessor of ACE in constraint programming. As it was used in the foundational paper underpinning this master's thesis [42], we retained its implementation to provide a baseline for comparison with more recent solvers like Choco and ACE.

3.1.4 Picat

Picat is a logic-based multi-paradigm programming language developed for general purpose applications [66]. It was the best performing solver on the last edition of the XCSP competition [61]. It would be interesting to add it to the Ludii framework to compare it to others solvers on our specific instances. However since Picat is its own language it does not offer the possibility to create a java executable for solving XCSP instances, by concern of ease of use for the Ludii users we chose not to include it in the current version of the framework.

3.1.5 MaxiCP

The last solver that was considered during this master's thesis was the MaxiCP solver [67], a lightweigth solver used for research in the field of constraint programming. This solver is based on the MiniCP solver [68] which is an educational solver partially developed by the UCLouvain. As the MaxiCP implementation contains the solutions to the various exercises of the classes related to MiniCP, a public version of this solver is not currently available. For this reason we chose to not add it to the Ludii framework for the time being.

3.2 Structural changes to the framework

The implementation of the artificial intelligences capable of solving the puzzle problems necessitated changes to the structure of the Ludii framework. We created two new packages called csp.Solvers and csp.utils.
csp.Solvers contains one class for each constraint programming AI utilizable in the framework (In this case there is one AI for each solver presented earlier) and a class called Translator, which implements the problemAPI class from the XCSP modeler. The problemAPI is the class used by the XCSP modeler to construct a new problem. By implementing this class we can use the provided methods to produce our own problems.
The csp.utils package contains the java executable of all the solvers and the shell scripts to execute them on the XCSP instance, as well as the temporary XCSP file generated by the AI.
Moreover in the class *BaseBooleanFunction*, which is the abstract class used to implement all constraint used in puzzle games, we added a new method *addConstraint*, which has the

following signature:

```
        public void addConstraint(ProblemAPI translator, Context context, Var[] x);
```

This method takes the model, the puzzle context, and the variables affected by the constraint as parameters, and then adds the constraint to the XCSP model.

3.3 IA class and translator

In this section we will study in more details the implementation of the different IA classes and the translator class.

3.3.1 IA class

The abstract IA class available in the framework allows us to implement AI agents capable of playing games. Two methods defined in this class are of particular importance to us, *initAI* and *selectAction*.
initAI is called only once when the AI is created and effectuate preliminary computations. In the case of constraint programming the whole solution has to be computed in one step (we need to be sure that the sequence of moves that will be played will not finish in a dead-end). In this method we build the XCSP problem using the *Translator* class. We save that problem in an XML file and use a command line call to the correct java executable. The solver return an answer in the Json format, that we can in turn translate to an array where each element corresponds to a variable of the puzzle.
selectAction is called every time the agent wants to play a new move. It creates a list of non-attributed variables (i.e each cell or vertice where no move has already been played), select randomly one of them, create a new move which attributes the correct value to this variable, then return this move which is effectively played on the game. The random selection is a design choice that allow to display the solution in an aesthetically pleasing way, we could also display the solution in the natural reading order, or in the order in which the variables were attributed.

3.3.2 Translator

The translator class is used for the translation of the ludemic representation of the game to an XCSP problem. It creates one variable in the XCSP problem for each constrained site in the puzzle, then restrict the domain of this variable to the range specific to the game. Then it applies unary constraints to all elements present in the starting rules, for example all numbers already present at the start of the problem in a Sudoku.
Then, the translator get all constraints from the game description and add all of them to a queue. It iterates over all elements of the queue and for each one of them it adds it to the XCSP problem. The use of a queue allows us to add more constraints to it if necessary. One case where it is necessary is when applying the constraint *and*, a constraint that take two other constraints and impose that the two of them are satisfied. To add this *and* to the XCSP problem, we simply add its two arguments to the queue, where there are in turn added to the XCSP instance.

3.4 Implementation of the different constraint

In this section we explain how the different constraints were implemented in Ludii. For each constraint we will focus on the most distinctive part of the implementation.

3.4.1 AllDifferent

Their is a simple relation between the *AllDifferent* ludeme and its translation in XCSP. We only have to state that that all the variables in a specific region have different values like this:

```
<allDifferent> x[0] x[9] x[18] x[27] x[36] x[45] x[54] x[63] x[72] </allDifferent>
```

However, the implementation of this constraint is non trivial due to the number of ways regions can be given in Ludii.

- The region can be defined as a list of sites, which can be directly translated to XCSP variables.
- The region can be defined as a *regionTypeStatic*, static regions are regions calculated at the instantiation of the game, such as columns or line, in order to avoid redundant computation during the execution of the game.
- The region can be defined by a *regionFunction*, which is a class that is used to define region. This class contains a method *eval()*, that given the correct *context* returns a *Region*, which contains a *site()* method which return the sites in the region as an array of integers.
- The constraint can be applied to all region with a specific name, in this case we iterate over all regions and compare their name to the constraint and select the corrects ones. Different regions can have the same name, this is useful when we want different constraints to apply to different sets of regions. For example in some game we want all "walls" to behave in a specific way and all "lines" to behave in an other way.
- Sometimes the region to which the constraint must be applied is not stated in the constraint, in this case we should apply the constraint to all regions on the board, including static regions defined earlier.

Once we have managed to obtain the indices of the constrained variables, the construction of the constraint in XCSP is straightforward, we simply need to convert the indices of cells to the XCSP variables and create a *AllDifferent* constraint in the model. Sometimes a specific value should not be taken into account in this constraint, for example we sometimes want every value in a region to be different or equal to zero. Thankfully, both Ludii and XCSP allow us to introduce one more argument in the constraint to state this exception.

3.4.2 IsSum

The constraint *IsSum* imposes that the sum of elements in a region is equal to a specific value. The way to gather all variables utilized is fairly similar to the constraint AllDifferent. The value of the sum can either be defined as an integer or as an hint. An hint is a specific way to define a region, with a collection of sites and a list of value, for example in Killer Sudoku[2], hints are defined in this way:

```
    (hint {0 9 18 27} 27)
```

Meaning that we create a region with the cells 0,9,18,27 and an "hint value" of 27. In the description of the rules of the game the sum constraint is then expressed like this:

```
    (is Sum HintRegions (hint))
```

Which imposes that for each region in HintRegions (i.e for every hint), the sum of the elements must be equal to the hint specified for this region.

3.4.3 AtMost/AtLeast

AtLeast and *AtMost* have really similar implementation to the *isSum* constraint. For the sake of brevity, we might be tempted to define one single ludeme to manage all classical constraint in a sum (equal, less or equal, strictly less, more or equal, strictly more), by adding one more parameter to the *isSum* ludeme. Besides, it would be more similar to the way these constraints are expressed in XCSP.
However that would be detrimental to one core aspect of Ludii developed in section 2.1.4, which is that Ludii aims to keep the ludemic representation simple to understand even for people without a formal programming background.

3.4.4 IsCount

The *IsCount* ludeme imposes that the number of repetition of an element in a region is equal to a specified value. The *count* constraint in XCSP is a direct implementation of this rule.

3.4.5 ForAll

The *ForAll* ludeme is used when one constraint has to be applied either to all instance of a site type (cells, vertices or edges) or to all hints. We simply need to get all the instances of the site type or all the region containing hints from the game context, then iterate over them and add the constraint contained in *ForAll* with the context corresponding to the current site.

3.4.6 AllSites

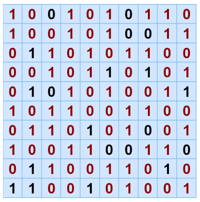
The ludeme AllSites apply a specific constraint to all the sites in a region defined by a *RegionFunction*. To add this constraint to the model, we evaluate the *RegionFunction* to obtain all the sites in this region, then we iterate over all these sites. For each one of these site, we add to the model the constraint specified in *AllSites* adapted to this site.

3.4.7 IsUnique

This ludeme states that for a set of regions of the same size each one of them has to differ by at least one value (i.e for each pair of region, there is at least one site such that the value

[2]A variant of Sudoku which add a constraint in the sums of specific regions

for this site is different in the two regions[3]). For now it is only used in Takuzu, a puzzle where we have to fill cells of a square grid while respecting some condition. Notably, one of these conditions is that all columns and all lines must be different. This is implemented in a quite straightforward way by using the XCSP constraint *AllDifferentList* which states that all lists given as parameters must differ by at least one element.



1	0	0	1	0	1	0	1	1	0
1	0	0	1	0	1	0	0	1	1
0	1	1	0	1	0	1	1	0	0
0	0	1	0	1	1	0	1	0	1
0	1	0	1	0	1	0	0	1	1
1	0	1	1	0	0	1	1	0	0
0	1	1	0	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1	0
0	1	1	0	0	1	1	0	1	0
1	1	0	0	1	0	1	0	0	1

Figure 3.1: A complete grid of Takuzu

3.4.8 IsTilesComplete

This constraint imposes that given some regions (composed of cells) each one is either full (each cell is gray) or completely empty. The way we chose to implement this constraint was by using a variant of the sum constraint in each region. We state that the sum of the values of the variables in the region must be either equal to zero or to the size of the region. Practically, this is written in XCSP as:

```
<sum>
    <list> x[3] x[7] </list>
    <condition> (in,{0,2}) </condition>
</sum>
```

Interestingly, as this constraint is used in only one game[4] and can be summarize easily as a compound of other constraints (The sum on the region is either the size of the region or zero), we could in future works try to implement *IsTilesComplete* as a ludemplex[5] in order to restrict the number of existing ludemes used in the construction of logic puzzles.

3.4.9 And

The *and* ludeme employs the queue structure utilized to store the constraints to add its two arguments of the *and* to the queue. It could be interesting to extends this concept to the implementation of *ForAll* and *AllSites* in order to simplify the code and allow a greater generality of the code.

3.4.10 IsMatch

As a reminder the IsMatch constraint, used mainly in the variants of the Nonogram (which rules can be found in section 2.1.6), imposes a constraint on the pattern of grayed cells in a

[3]Of course, the sites of these regions must be ordered in a specific way for this constraint to make sense, in the case of Takuzu it applies to column and line so the order is the usual reading direction, however we can easily imagine other geometries where this would not be as trivial.

[4]Tilepaint

[5]LudemPlexes are compound structures of ludemes that aims to facilitate the comprehension of the representation and increase reusability of the code.

specific line. The numbers on the left of that line (or the top of that column) indicate the consecutive groups of shaded squares, separated by at least one empty square.
For example, the hint below imposes that the line contains any number of whites cells, followed by four consecutive black cells, at least one white cells, two consecutive black cells, at least one white cells, two consecutive black cells and any number of white cells.

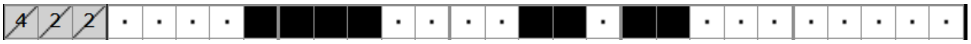


Figure 3.2: A solved line from a Nonogram

The astute reader has already noticed that given the previous definition of the problem, the translation to a regular expression is straightforward. Regular expressions are sequences of characters that specify a match pattern [69]. A match pattern is a tool to concisely and formally check that a given sequence of token respect its rules.
The aforementioned hint can therefore be formally expressed as the regular expression: $0^*1\{4\}0+1\{2\}0+1\{2\}0^*$ [6]. One interesting property of regular expression is that they can be transformed into non-deterministic finite automaton [70], and as we have shown in 2.2.4 we can construct constraints in XCSP that ensure a sequence of variables can be recognized by an specific automaton. Thus, the previous constraint can be represented like this in this XCSP instance:

```
    <regular>
            <list> x[0..4] </list>
            <transitions>
            (0,0,0)(0,1,1) (1,0,11)(1,1,2)
            (2,0,11)(2,1,3) (3,0,11)(3,1,4)
            (4,0,5)(4,1,11) (5,0,5)(5,1,6)
            (6,0,11)(6,1,7) (7,0,8)(7,1,11)
            (8,0,8)(8,1,9) (9,0,11)(9,1,10)
            (10,0,10)(10,0,11)
            </transitions>
            <start> 0 </start>
            <final> 10 </final>
        </regular>
```

In this automaton:

- The states 0 is the initial state,
- States 1,2,3,4,6,7,9 and 10 correspond to states for which the last value on which we iterated was a black block.
- States 5 and 8 are states in which we just iterated in one white block after a group of black cells.
- The state 10 is a terminal state. The constraint is satisfied if the automaton finish on this state.

[6]zero or more 0 followed by exactly four 1, followed by one or more 0, followed by exactly two 1, followed by one or more 0, followed by exactly two 1, followed by zero or more 0

- The state 11 is a degenerated state which is not terminal (i.e if the finite automaton finish in the state 11, the constraint is not respected). We reach this state if we "cut" a sequence of black cells that can not be separated, if we extends too much a sequence of black cells, or if we add a fourth group of black cells.

Equivalently we can show the corresponding finite automaton:

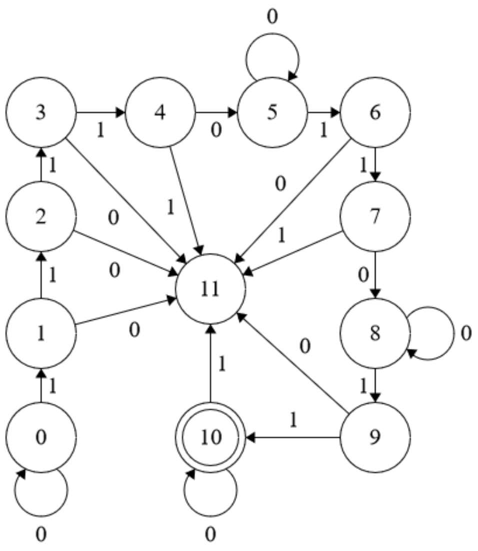
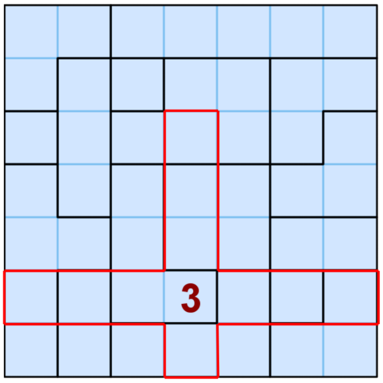


Figure 3.3: The finite automaton corresponding to our IsMatch constraint

3.4.11 IsDistinct

The *IsDistinct* ludeme is only used in RippleEffect. In this game we have to fill a grid with numbers, one of the conditions to respect being that if a cell is filled with one number, no other orthogonal cell orthogonally at a distance inferior or equal to the number in the first cell can have the same value. In the following board none of the cells surrounded in red can have the value 3:



Due to the nature of this constraint, we cannot know the variables that will be used in it before calculating the solution. XCSP does not provide a standard way to implement this type of dynamic constraint. We need to circumvent this limitation by combining simpler available constraints.
One way to do so is to use the extension constraint as described in section 2.2.4.

- For each cell we compute regions of all possible size (One for each possible value of the cell).
- For each one of these regions, we iterate over all cells and create a new extension constraint stating that the pair of cell <current cell, center cell> can not have the value <size of the region, size of the region>.

The constraint in one of these pair of cell would be expressed like this in the XCSP model:

```
    <extension>
        <list> x[10] x[12] </list>
        <conflicts> (2,2) </conflicts>
    </extension>
```

In this case, the extension states that the variables x[10] and x[12] can not both have the value 2.
We can see that this method produce a huge number of constraints, in our example the bottom left cell produce 34 constraints. However this does not seem to be a significant problem for the resolution speed, ACE can solve the 17x17 problem in a little more than 2 seconds, despite the XCSP model producing 4229 constraints. Obviously there are far more efficient ways to represent the ludeme in this specific game, we chose this particular implementation in order to conserve the generality of this constraint with different board geometries or region shapes.

4 Results

4.1 Correctness

We did not manage to solve all puzzles currently implemented in Ludii, you can see below a table showing all the games and their current status:

Solved games			Unsolved games	
Akari	Anti-knight Sudoku	Asterisk Sudoku	Big Tour	Hashiwokakero
Buraitoraito	Butterfly Sudoku	Center Dot Sudoku	Hitori	Usowan[1]
Color Nonogram	Fill A Pix	Flower Sudoku	Masyu	SlitherLink
Futoshiki	Girandola	Hexagonal Nonogram		
Hoshi	Jigsaw	Kakuro		
Kazaguruma	Killer Sudoku	Latin Square		
Magic hexagon	Magic square	N Queens		
Nonogram	Ripple effect	Samurai Sudoku		
Sohei Sudoku	Squaro	Sudoku		
Sudoku DG	Sudoku mine	Sudoku X		
Sujiken	Takuzu	TilePaint		
Tridoku	Windoku			

4.1.1 Unsolved problems

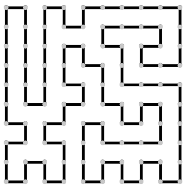
Big Tour

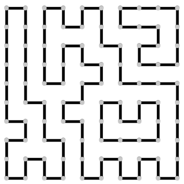
Big tour is one of the simplest graph puzzle, it requires the player to draw one single large loop that passes through each vertices of the graph. The constraints implied in this problem are fairly simple to model, except the one stating that the edges must form one single loop. This imposes that all edges must be connected, in the ludemic implementation the ludemplex *"OnlyOneWay"* is used in the end conditions to ensure that the edges form a single loop. This ludemplex is defined as follow:

[1]While testing this game we realized it was incorrectly implemented, making its resolution outright impossible to achieve.

```
    (define "OnlyOneWay"
              (and (is Solved) (= (count Groups Edge) 1))
    )
```

As we can see, it states that for the puzzle to be considered solved the edges have to form a single group[2], which in the case of the edges in a graph is equivalent to stating that they must be connected.
There exist algorithms to solve this type of problems [71][72], however constraint programming is not often used for this class of problems and commercial solvers lack the global constraint necessary to easily implement them. Thus we were not able to solve problems involving connectivity constraint. This issue is further discussed in section 5.1.


(a) Correctly solved Big Tour


(b) Solution without the connectivity

SlitherLink and Masyu

These two games have very similar rules to Big Tour, with the exception that the loop does not have to cross every vertex.
In the case of SlitherLink some cells contain an hint that indicates the number of neighboring cells that must be used. In Masyu, some vertices contain hint to indicate if the loop must turn or continue straight at that point.
All of those constraint are fairly straightforward to implement, but the connectivity constraint once again hinder the complete resolution of the games.

Hashiwokakero

In Hashiwokakero, we have to link some islands with bridges while respecting the following conditions:

- Bridges cannot cross each other.
- Bridges are drawn orthogonally.
- The maximum number of bridges between two island is two.
- The numbers on the islands indicate the number of bridges connected to this island.
- All islands must be connected.

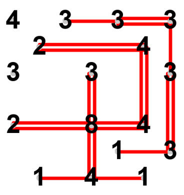


Figure 4.1: A half solved Kakuro

[2]The ludeme *(count Groups)* count the number of contiguous groups of the specified element

Once again the constraint that prevent us to solve this problem is the one stating that all islands must be connected. It notably uses the same ludeme *"OnlyOneWay"* than the previous puzzles.

Hitori

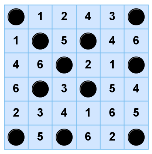


Figure 4.2: A solved Hitori

In Hitori, the player receives a grid with numbers on it. To solve the puzzle he has to place black pieces on some cells in order to keep all different numbers on each line and column. No two black cells can be placed side by side and all the cells without pieces must be contiguous.
The difficulty of solving this game using constraint programming resides in the connectivity constraint on white cells. It is ensured by the ludeme *(is Connex 0)* that states all cells containing a 0 (i.e all empty cell) must be connected. This is the only puzzle in which this ludeme is used but while trying to produce an XCSP equivalent we encounter the same difficulties than the *"OnlyOneWay"* ludeme, which is logical since those two constraints are conceptually equivalent.

Usowan

In this game, the player has to color cells such that each cell with a number indicates the number of neighboring colored cells (cells with numbers cannot be connected). Colored cells cannot be connected and all white cells must be orthogonally connected. However, in each rectangle bordered by bold lines, there is exactly one hint which is *a lie*, and that we must not respect.

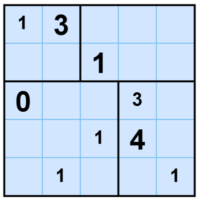


Figure 4.3: An empty grid of Usowan

We were not able to solve this game, partly because it was inadequately implemented in Ludii. The designer of the game made a distinction between "true" and "fake" hints while designing the game. Because good puzzles games should have only one solution this does not hinder the ability of an human player to play the game. Similarly, tests executes correctly. However this design choice poses some problems:

- This solution can only be used when the designer know beforehand the solution, as it needs to manually distinguish between good and bad hints.

- It cannot not be used if the puzzle has more than one valid solution, since one specific solution is expected by the framework, it would invalidate a good answer. A correctly design puzzle should have only one solution, but we can not expect that it will always be the case.
- Artificial intelligences cannot be applied fairly to these problems. As they have access to the underlying representation of the problems, separating the true and false hints becomes trivially simple and remove any significant meaning of "Solving" the puzzle.

This emphasize the need to design problems in a "neutral" way. The verification of the validity of the problems should never be depends of a solution previously known by the designer of the puzzle. This may seems obvious from the point of view of someone trying to develop a solver agent. However as seen previously it may not be directly evident for someone creating a new game. Even more so if he has no computer sciences background.

4.1.2 Generalization to new problems

One of the main objective of our agent was to offer a generic solver capable of generalizing resolution methods to new problems. We were able to achieve this objective, at least partially. In order to prove this, we tried implementing new games in Ludii without further modification of the agent, to test the generalization capabilities of our solver. One such game is *Thermometer Sudoku*, a variation of the Sudoku which add a constraint on "thermometers" stating that the numbers should be in increasing order starting from the bulb of the thermometer. One example of this puzzle is present in image 4.4.

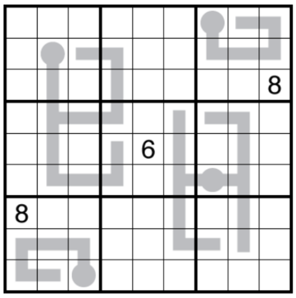


Figure 4.4: A thermometer Sudoku

We can model this problem by combining the classical rules of the Sudoku with the < ludeme used in Futoshiki[3]. Upon calling our AI on this game, it produces the solution in image 4.5. The graphical elements for this particular games are not present in Ludii, which explain the peculiar look of the solved puzzle, but with further inspection one can convince himself that the solution indeed respects the constraints stated in the initial problem.

Generally, we should be able to solve all puzzles which rules are already used in solved games. We should also be able to solve all different geometries and sizes of existing games, this is particularly important for the Sudoku, who possess a great number of variants.

[3]This ludeme ensure that a variable has a lower value than the other

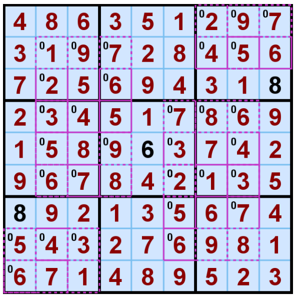


Figure 4.5: The solved Thermometer Sudoku

However after examining the games in Nikoli we can see that many of them are not implemented in Ludii and therefore are not solvable using our agent. Obviously, most of the games representable using the existing ludemes were already added by [44]. Our solver thus lack generality in the sens that there exist many puzzles that it can't solve.
It stands to reason that puzzles enthusiastic will seek original games that varies from known constraints and usual patterns, Nikoli's website claim that each puzzle has "a totally different “flavor” and freshness" [3]. This tendency to seek new original rules in puzzle reduces the generalization potential of our agent. Many games in Nikoli feature an unique constraint that is not shared by any other. A good example of such a constraint is found in ShakaShaka, where the player has to add black triangles to a grid in a way that creates rectangles (which can be "tilted"). No other puzzle features a rule similar to this one, it's probable that we will have to create a custom ludeme specifically for it.

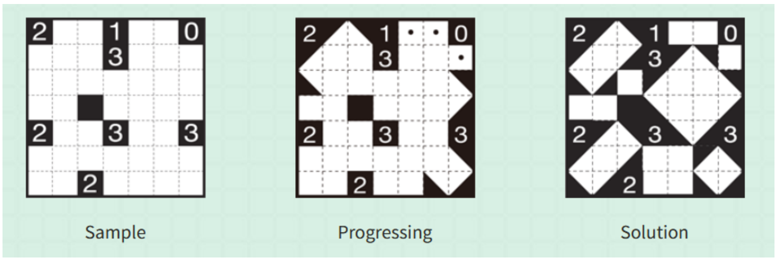


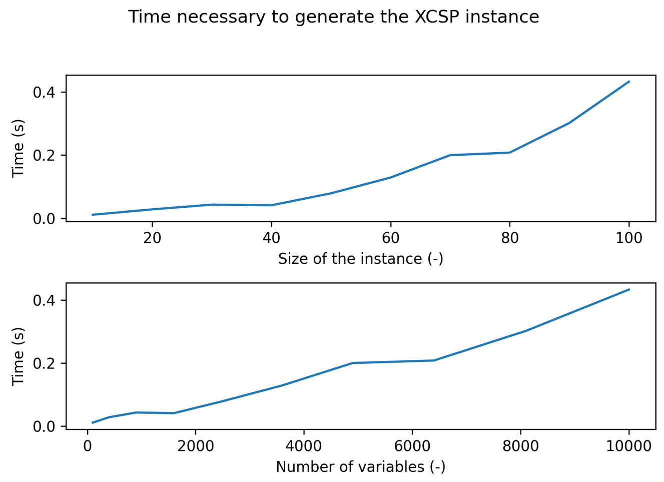
Figure 4.6: A game of ShakaShaka

4.2 Speed

In this section we will observe the time taken by the solver to translate a game in the XCSP format and the time taken by the solver to produce an answer. We will compare different solvers to determine if their is a significant difference of performance between them . If it is the case, we will try to determine which criterion determine the best solver for a specific puzzle.

4.2.1 Generation of the XCSP problem

The first step in the resolution of the puzzle is to translate the Ludii representation to an XCSP instance. We measure the time taken by the XCSP modeler to model the puzzle and write its representation in the *Translator.xml* file.
We chose three different problems on which to take measurements. First, the N-Queens, a common benchmark when comparing resolution speeds for constraint programming solvers [73], and which relative simplicity[4] allows Ludii to run large instances efficiently. Second, the Ripple effect puzzle, characterized by a huge number of constraints in its XCSP representation. Finally, the Nonogram, which incorporates non-standard constraint that may require more time to add to the instance.
In the following graph we show the time necessary to produce the XCSP instance depending on the size of a N-Queens problems (The size of one side of the board) and the number of variables:



We can see here that even for problem of very large size, with up to 10 000 variables and 600 constraints, the generation of the XCSP instance takes less than half a second. We did not conduct experiments with larger board sizes since Ludii is unable to support them.

Next, we study the time necessary for the agent to produce an XCSP representation of the Nonogram puzzle. This representation uses regular constraints which are implemented as described in section 3.4.10, and that we could expect to be more computationally expensive to construct. However, as we can see in the table 4.1, the time taken to create the XCSP model remains relatively short, even for large instances. We will see later how this delay in the resolution is negligible when compared to the resolution time and the

[4]In the N-Queens problem, the player is given a board of size N×N and must place N chess queens such that no two queens can attack each other.

Size of the puzzle	2	5	10	15	20
Time (ms)	3	6	12	12	27

Table 4.1: Creation of the XCSP instance for the Nonogram puzzle

limitations induced by the Ludii framework itself.

Finally, we examine the Ripple Effect puzzle and the time required to produce an instance of the XCSP problem. This problem is interesting in that it produces an exceptional number of constraints. For example, in the largest instance available on Ludii, which is a square of size 17, the model enforces 41 154 constraints. In the table 4.2 we can see for each available size of Ripple Effect the total number of variables created, the number of constraints, the time taken to create the XCSP instance and the number of constraints created by milliseconds of runtime. This latter metric is obtained by dividing the total number of constraints by the time necessary to produce them.

Size of the instance (N*N)	Variables	Constraints	Time (ms)	Constraints by millisecond
6	36	1368	7.4	184.8
8	64	2611	11.6	225.1
10	100	6288	36	174.7
17	289	41154	174	236.5

Table 4.2: Creation of the XCSP instance for the Ripple Effect puzzle

Interestingly, we see that the number of constraints produced per second is fairly constant whatever the size of the puzzle. Although we lack the quantity of instances to analyze in details the evolution of the number of constraints, it is evident that it does not evolve in a linear fashion. Thus, by increasing the size of the puzzle we will reach unacceptable computation time on large instances. The resolution of those instances would require a more efficient way to implement the rules of this puzzle.
That said, the computation times obtained for the available instances are reasonable given that the 17x17 case is the largest Ripple Effect that we could find. This game being chosen specifically because of the huge number of constraints generated[5], this demonstrates that the current implementation can handle moderately large puzzles effectively.

4.2.2 Resolution Speed

In the following table, we present the computation time required to solve instances of various problems using the Choco and ACE solvers. The computational overhead of Ludii can sometimes hinder the ability to produce solutions within a reasonable time frame as described in [44], independently of the solvers' performance. Therefore, for each solver, we compare the resolution speed within the Ludii framework against directly invoking the solver on the XCSP instance using external commands, without using Ludii's architecture.

[5]By way of comparaison, the 16x16 sudoku only produce 48 constraints, the 100x100 NQueens 602 constraints, the 16x16 TilePaint 120

When a game could not be solved in a reasonable time frame (We stopped the computation after 5 minutes when the solver could not find a solution), we denoted it with an "-".

	AbsCon		ACE		Choco		Best
	Ludii	No Ludii	Ludii	No Ludii	Ludii	No Ludii	solver
NQueens 10*10	0.591	0.536	0.92	0.781	9.473	9.384	AbsCon
NQueens 15*15	14.731	14.642	-	-	-	-	AbsCon
NQueens 20*20	-	-	-	-	-	-	-
Nonogram 5*5	0.407	0.375	0.693	0.578	0.438	0.389	AbsCon
Nonogram 10*10	0.449	0.424	0.718	0.645	0.484	0.407	AbsCon
Nonogram 15*15	0.497	0.427	0.768	0.625	0.498	0.440	AbsCon
Nonogram 20*20[6]	-	0.44	-	0.696	-	0.460	AbsCon
Latin Square 10*10	0.425	0.374	0.694	0.546	0.38	0.344	Choco
Latin Square 20*20	0.561	0.490	0.78	0.671	0.448	0.406	Choco
Latin Square 30*30	0.806	0.696	0.918	0.828	0.589	0.536	Choco
Latin Square 50*50	3.22	3.249	2.236	2.197	1.412	1.433	Choco
Latin Square 100*100	-	85.215	-	44.902	-	-	ACE
Ripple Effect 10*10	0.845	0.760	1.08	0.968	0.831	0.764	Choco
Ripple Effect 17*17	2.06	2.031	2.284	1.984	2.043	1.977	Choco
Akari 7*7	0.424	0.375	0.701	0.593	0.418	0.357	Choco
Akari 14*14	0.525	0.471	0.804	0.671	0.356	0.409	Choco
Akari 25*25	0.579	0.540	0.895	0.781	0.516	0.453	Choco
Samurai Sudoku	0.505	0.469	0.795	0.687	0.476	0.424	Choco
Buraitoraito	0.423	0.376	0.707	0.578	0.422	0.376	Choco
TilePaint 16*16	0.458	0.409	0.787	0.656	0.407	0.361	Choco

Table 4.3: Comparison of the solvers on a subset of the puzzles

Comparison of the solvers

When comparing the time taken by the different solvers on the puzzle we see that generally Choco performs slightly better than AbsCon, with ACE performing a bit worse. Nonetheless the results on two particular puzzles stand out: Choco performs exceptionally badly on the NQueens problems, and ACE performs exceptionally great on the Latin Square problem. Of course it is expected that solvers will behave differently on different problems, however it is compelling that the problem on which ACE performs the best is the one containing the largest number of variables.
One important thing to consider when comparing the solvers is that the ludemic version of the problems was not necessarily designed to be easily solved by constraint programming. For example, the constraints on the NQueens problems of size N simply state that the number of queens for all lines, columns and diagonals is *at Most* 1 and that the board contains N queens. It is possible that a reformulation of the rules of the game could speed up the resolution process, or change the best performing of solver.

The fact that the AbsCon solver performed better than its successor on almost all instances is quite surprising. We used the solvers in a generic fashion without fine tuning them for specific problems, it's possible that further improvements on the matter would

[6]The 20 by 20 Nonogram that was present in Ludii is not satisfiable, the time taken by the solver is the elapsed time taken by the solver to prove that the problem is not satisfiable. This was probably not realized during its implementation since Lit could not be tested by the Framework.

show better performances for ACE. Besides, problems with more variables seem to be more easily handled by the ACE solver, but we would need more large instances in Ludii to demonstrate it formally.

All things considered, it seems that different solvers will be best for specific problems and problems sizes. The chosen approach is thus to configure Ludii to provide for each problem the solver that is best suited. This solver being empirically determined, it harms the generalization possibility needed in Ludii. What's more, we will need to do the tests again each time we update one solver or add a new one. Additional research will be useful in identifying *a priori* the conditions under which a solver performs effectively.

4.2.3 Limiting factor in resolution

It's important to identify the limiting factor in the resolution of problems in Ludii to focus correctly our effort to improve it. We have identified three possibles reasons for which game could not be solved in a reasonable time by Ludii:

- Generation of the XCSP instance: As seen in section 4.2.1 the generation of the XCSP instance stays fairly efficient, even for problems with large sizes and high number of constraints. It could however be a future problem for games with exceptional number of constraints when reaching large sizes, such as Ripple Effect.
- Solving process: We saw in table 4.3 that the resolution process take a long time on the larger problems. This is not an issue on most human-sized puzzles, but prevents us to solve big instances of games like N-queens or Magic Quare and could lead to some problems in the futur.
- Processing by Ludii: Some games are not currently run efficiently by Ludii, they are not loaded correctly or are too slow to display the solution, even if it is correctly computed. For example, the Nonogram of size 20 only allow for one move to be played every 1208 seconds. This is discussed in details in [44], where the author identify two possible causes for this inefficiency.
  The first limitation is the number of pre-calcutions performed by Ludii while loading the game, limiting the size of the board. These pre-computations reduce the time necessary to calculate potential moves but increase the loading time of the game, when the board is too large this pre-computation becomes too expensive and completely block the framework. This explain the inability to run tests on instances of N Queens and Latin Square of sizes superior to 100.
  The second limitation in the speed of the framework is the computation of some constraints. This mainly affects the Nonogram, for which the current code compute all the possible values for each line and each column, and for each move played checks that the resulting array belongs to one of these possible values. This is computationally expensive and slow down the execution of the game to the point it is unplayable.

5 Future works

5.1 Implementation of the last constraints

As discussed in the previous chapter of this thesis, we were not able to implement constraints on the connectivity of a solution. This prevents us from solving some puzzles in Ludii, and will prevent us from solving many other puzzles that are not yet in Ludii. The main problem with this constraint seems to be that we cannot apply a constraint on a subset of vertices to enforce connectivity. Generally we can expect that we will struggle to produce an XCSP representation of every games where the constraints can't be expressed on subset of variables.
There are possible solutions to this problem that are worth exploring:

- The use of other programming paradigms in conjunction with constraint programming. Once the problem is formulated as a graph problem, classical resolution methods can be applied. Obviously, it would require significant changes to the internal representation of the puzzles.
- The creation of custom global variables. This would imply to design a specific constraint that enforce connectivity and modify the XCSP standard as well as the solver to take into account this new constraint. This has already been done, for example the constraint programming solver used in [58] implement a **CONNECTED()** constraint which imposes edge connectivity.

5.2 Extension to more games

In Pierre Accou's master thesis [44] the author tried to implement the greatest number of games from both Nikoli and Cross+A, however, many puzzles remain non-playable. We can see in the graph 5.1 the proportion of games present in Ludii for both Nikoli and Cross+A.

In order to try and solve them using constraint programming, we should strive to add more of those puzzles to Ludii. It is not easy to exhaustively describe all the puzzles that have yet to be implemented and their difficulties. However we can provide some main

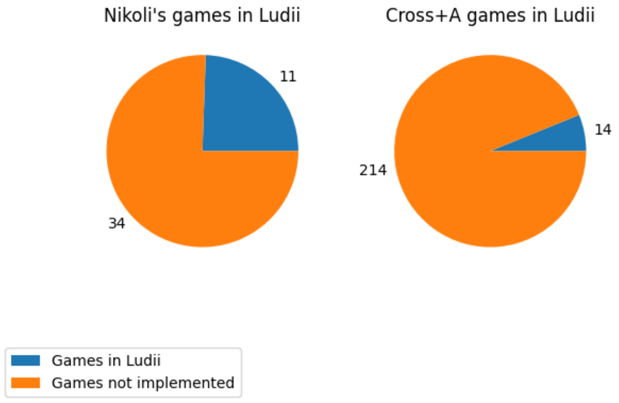


Figure 5.1

categories of puzzle that share common characteristics not seen in puzzle already studied.

The first of those categories require the player to delimit regions of specific sizes and shapes by coloring edges[1]. One example of this category is shown in figure 5.4. The way to implement this problem in Ludii is not immediately obvious, conceptually the game require the player to divide the board on different regions, but the ability to "toggle" vertices suggests that they should be the variables in the constraint solver.

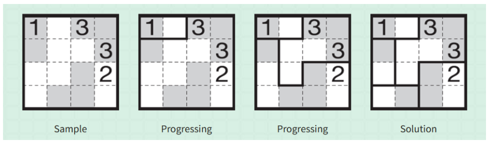


Figure 5.2: Resolution of a Double Choco puzzle

The second category is composed of puzzle like Yajilin, Herugolf or Suraromu. These puzzles require the player to construct a path that "moves" in a certain way. For example in the Suraromu problem visible in figure 5.3, we have to construct a loop that "passes" through each door, the numbers on doors indicating that the loop must cross them in a specific sequence (The door with the number 2 must be the second door crosses). While this notion of "crossing" a door is fairly intuitive for human player, to the best of our knowledge no XCSP constraint representing this type of limitation exist at the moment.

[1]Including Double Choco, Jemini, Neibadomino, Yokibunkatsu among many others

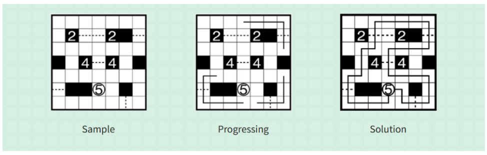


Figure 5.3: Resolution of a Suraromu puzzle

The final category of games that we were able to identify was the one containing puzzles like Shakashaka, Shikaku or Tentai Show, where the player has to modify the board in order to create specific properties for each region of the grid. In Shakashaka and Shikaku the board must be modified to create regions of square or rectangular shapes. In Tentai Show, he has to create regions that are symmetrical with respect to some pre-given point of reference. These constraint cannot be expressed by local constraints in XCSP and will probably require the creation of some custom global constraint.

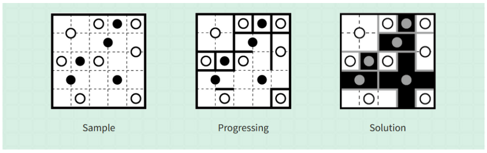


Figure 5.4: Resolution process of a Tentai show, notice how all pieces are symmetrical with respect to the whites and black dots.

5.3 Improvement of Ludii speed

As shown in Chapter 4, a significant obstacle to the resolution of large instances of some puzzles is the computational speed of the Ludii framework itself.
This issue is particularly evident with the Nonogram, which struggles to handle even moderately sized puzzles compared to those in specialized collections [74]. The critical path to providing an efficient Nonogram solver in Ludii lies in improving the framework itself and enhancing its computational speed.
Part of the problem is linked to Ludii's move-validation mechanism, which checks that moves played are legal. Each time an agent try to apply a new move to the board, Ludii verify that is doesn't violate any constraint, and if it does, it prevents the user to play this move. In the case of the Nonogram, Ludii calculates all possible values for each line and column (Depending on the hint) to check if the current states of the variables is still coherent with the hint. This is not really efficient as it requires the creations of a huge number of candidates for each constraint, preventing the puzzle from working for larger problem sizes.
One other way Ludii could be improved is by reducing the quantity of pre-computation on boards of very large sizes, such as the ones used for the N-Queens or Magic Square puzzles. The impact of this pre-computation is clearly shown in [44], where a simple 25*25 N-Queens puzzle takes more than 6 seconds to load in Ludii. A path to improve this is to analyze the description of the game in order to compute only the information relevant to

the game. When running an instance of the Magic Square, Ludii should not compute all the possible knight moves in the board. One added benefit of this amelioration is that it would equally improve all other games present in Ludii, even the ones that are not logic puzzles.

5.4 Further improvements for users

The current solver allows the user to solve a problem and input its solution in Ludii. This is helpful to find or verify the solution to a puzzle. We could add many more useful features for both players and game designers.
A human player using the Ludii interface could be blocked on a specific puzzle, an instead of outright receiving the solution, he could want to receive an hint to continue progressing in a more natural way. A straightforward way to assist the user is to indicate if there's a mistake in their current puzzle resolution. This can be done by running a solver on the current state of the board to check if a valid solution exists. If the user requires more assistance we can trivially compute the solution and reveal the value of one random site. However, this naive solution, while easy to implement, is probably of little interest to a knowledgeable player. Indeed, a good puzzle problem should be solvable using mainly inductive reasoning and involve very little guessing[2]. A good hint, rather than blindly pushing the player in a random direction should identify a subset of the problem that can be reasonably solved by an human player.
On simple instances of puzzles we can identify the sites with the smallest domain, but this approach will not work on more complex puzzle. Some papers have tried to produce solvers capable of identifying known patterns for specific games [75] [76], but these methods are limited to specific puzzle with handcrafted techniques. (As an order of magnitude, there exists more than fifty different strategies used by human players for the sole Sudoku[77]). The most promising approach to increase explainability in general game playing is be described in [78], in this paper the authors use *Minimal Unsatisfiable Subsets* (MUSes) to generate a sequence of small reasoning steps, each with an explanation understandable for a human player.

A game designer using Ludii could want to design new instances of known puzzles or to create entirely new puzzles. To create new instances for known puzzle it would be interesting to explore the automatization of the creation process, as discussed earlier designing **interesting** instances is far from easy, and would need to take into account the solving methods available to the player, for example by using the MUSes described above. Moreover the creation of hand-made new instances would be greatly facilitated by a graphical interface, the current process of writing all hints by referencing the indexes of the affected sites being quite cumbersome. Finally, the generation of new puzzle using evolution and language models as described in [40] could yield promising results.

[2]This is probably why Nikoli doesn't use procedurally generated puzzles, who may require the use of a lot of "backtracking" methods, which human players don't enjoy.

6 Conclusion

The objective of this master's thesis was to develop an agent capable of solving every logic puzzle. To achieve this, we utilized the Ludii framework and its Ludemic description of games to produce standardized XCSP representations of the problems. These representations enabled us to use constraint programming solvers, such as ACE and Choco, to compute solutions, which were subsequently integrated back into the Ludii framework.

Our approach successfully solved a wide variety of puzzles already implemented in Ludii. Furthermore, we demonstrated that our implementation was capable of solving new games by adding them to Ludii and solving them without further modification of the code. We also showed that the creation of the XCSP instance introduced minimal delay, enabling the utilization of this approach for real-time use by human players. Similarly, even if the resolution process does not reach the performances of custom designed implementations and propagation strategies tailored for specific problems, it was fast enough for the resolution of typical human-sized puzzles.

Despite these achievements, significant challenges remain for Ludii to be able to solve all logic puzzles. The Ludii framework lacks the efficiency to run very large instances due to the amount of unnecessary pre-computing realized on the board. Some puzzles, such as Nonogram, require heavy computation even for relatively small instances, which hinders the playability of the game and the presentation of the solution to the user.

Moreover, many games are yet to be implemented in Ludii, more than 90% of the puzzles present in the database used as references are not part of the framework. We cannot guarantee that those puzzles would be solvable by the agent in its current state. In fact, the presence of constraints not currently implemented in Ludii for some of these games suggests that modifications to the agent would be required to solve them.

Finally and most importantly, some games already present in Ludii remain unsolved, particularly puzzles which incorporate connectivity constraints. These constraints pose unique challenges due to their complexity and the lack of XCSP support, highlighting the need for further research on this topic.

Ultimately, this work demonstrates that the creation of a generic solver for deduction puzzles is indeed possible. By combining the Ludii framework with the XCSP formalism and existing constraint programming solvers, we have laid a solid foundation for future research on this topic.

Bibliography

[1] HandWiki. Mathematics of sudoku. *Encyclopedia. Web*, 10 2022.

[2] Takayuki YATO and Takahiro SETA. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A, 05 2003.

[3] NIKOLI Co, Sep 2021. `https://www.nikoli.co.jp/en/puzzles/` [Accessed: Oct. 2024].

[4] Sergey KUTASOV and Ilya MOROZOV. Logic puzzles. `https://www.cross-plus-a.com/fr/puzzles.htm` [Accessed: Oct. 2024].

[5] Simon ANTHONY and Mark GOODLIFE. Cracking the cryptic. `https://www.youtube.com/@CrackingTheCryptic` [Accessed: Oct. 2024].

[6] Édouard LUCAS. *Récréations mathématiques*, volume 3, page 58. Gallica, 1892.

[7] Lianne Hufkens and Cameron Browne. A functional taxonomy of logic puzzles. pages 1–4, 08 2019.

[8] Cracking The Cryptic. A sudoku of astonishingly beautiful lines. `https://www.youtube.com/watch?v=aE-TiTDHNpk` [Accessed: Oct. 2024].

[9] Cracking The Cryptic. The magic you can do with the factors of 12. `https://www.youtube.com/watch?v=zJdyKjsliGI` [Accessed: Oct. 2024].

[10] Cracking The Cryptic. Why hand made? `https://www.nikoli.co.jp/en/puzzles/sudoku/why_hand_made/` [Accessed: Oct. 2024].

[11] Alex Bellos. Inside japan's cult-favorite puzzle laboratory, 2022. `https://www.atlasobscura.com/articles/nikoli-puzzle-communication` [Accessed: Oct. 2024].

[12] Eric Piette, Dennis J.N.J. Soemers, Matthew Stephenson, Chiara F. Sironi, Mark H.M. Winands, and Cameron Browne. Ludii - the ludemic general game system. In Giuseppe De Giacomo, Alejandro Catala, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 : 24th European Conference on Artificial Intelligence*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418, Netherlands, 2020. IOS Press.

[13] Éric Piette, Cameron Browne, and Dennis J. N. J. Soemers. Ludii game logic guide. *arXiv:2101.02120*, 2022.

[14] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[15] Frederic Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3: An integrated format for benchmarking combinatorial constrained problems, 2024.

[16] Michael Genesereth and Michael Thielscher. *General Game Playing*. Morgan and Claypool, 2014.

[17] Michael Genesereth and Nathaniel Love. General game playing: Game description language specification. *Computer Science Department, Stanford University, Stanford, CA, USA, Tech. Rep*, pages 1–48, 2005.

[18] The International General Game Playing Competition — logic.stanford.edu. `http://logic.stanford.edu/ggp/readings/retrospective.html`. [Accessed 07-11-2024].

[19] Gamemaster - Gamechecker — ggp.stanford.edu. `http://ggp.stanford.edu/gamemaster/homepage/gametester.php`. [Accessed 07-11-2024].

[20] Éric Piette, Frederic Koriche, Sylvain Lagrue, and Sébastien Tabary. Woodstock : Un programme-joueur générique dirigé par les contraintes stochastiques. *Revue d intelligence artificielle*, 31, 06 2017.

[21] Frédéric Koriche, Sylvain Lagrue, Éric Piette, and Sébastien Tabary. General game playing with stochastic csp. *Constraints*, 21(1):95–114, August 2015.

[22] S. Schiffel and M. Thielscher. Representing and reasoning about the rules of general games with imperfect information. *Journal of Artificial Intelligence Research*, 49:171–206, February 2014.

[23] Michael Thielscher. Gdl-iii: A description language for epistemic general game playing. In *IJCAI*, pages 1276–1282, 2017.

[24] Chiara F. Sironi and Mark H. M. Winands. Optimizing propositional networks. In Tristan Cazenave, Mark H.M. Winands, Stefan Edelkamp, Stephan Schiffel, Michael Thielscher, and Julian Togelius, editors, *Computer Games*, pages 133–151, Cham, 2017. Springer International Publishing.

[25] Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, and Marek Szykuła. Regular boardgames, 2018.

[26] Jakub Kowalski, Jakub Sutowicz, and Marek Szykula. Regular boardgames. *CoRR*, abs/1706.02462, 2017.

[27] Jakub Kowalski, Radoslaw Miernik, Maksymilian Mika, Wojciech Pawlik, Jakub Sutowicz, Marek Szykula, and Andrzej Tkaczyk. Efficient reasoning in regular boardgames. In *2020 IEEE Conference on Games (CoG)*, volume abs 1908 9453, page 455–462. IEEE, August 2020.

[28] Achille Morenville, Éric Piette, and UCLouvain ICTEAM. Vers une approche polyvalente pour les jeux à information imparfaite sans connaissance de domaine. In *Plate-Forme d'Intelligence Artificielle-Rencontres des Jeunes Chercheurs en Intelligence Artificielle*, 2024.

[29] Dennis J.N.J. Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. The ludii game description language is universal. In *2024 IEEE Conference on Games (CoG)*, pages 1–8, 2024.

[30] Cameron Browne, Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, Michael Conrad, Walter Crist, Thierry Depaulis, Eddie Duggan, Fred Horn, Steven Kelk, Simon M. Lucas, João Pedro Neto, David Parlett, Abdallah Saffidine, Ulrich Schädler, Jorge Nuno Silva, Alex de Voogt, and Mark H. M. Winands. Foundations of digital archæoludology. *CoRR*, abs/1905.13516, 2019.

[31] Walter Crist, Éric Piette, Dennis Soemers, Matthew Stephenson, and Cameron Browne. *Computational Approaches for Recognising and Reconstructing Ancient Games: The Case of Ludus Latrunculorum.* 01 2022.

[32] Eric Piette, Lisa Rougetet, Walter Crist, Matthew Stephenson, Dennis Soemers, and Cameron Browne. A Ludii analysis of the French Military Game. In *XXIII Board Game Studies*, Paris (en ligne), France, April 2021.

[33] Éric Piette, Walter Crist, Dennis J.N.J. Soemers, Lisa Rougetet, Summer Courts, Tim Penn, and Achille Morenville. Gametable cost action: kickoff report. *ICGA Journal*, 46(1):11–27, September 2024.

[34] Dennis J.N.J. Soemers, Jakub Kowalski, Éric Piette, Achille Morenville, and Walter Crist. Gametable working group 1 meeting report on search, planning, learning, and explainability. *ICGA Journal*, 46(1):28–35, September 2024.

[35] Cameron Browne. Everything's a Ludeme Well, Almost Everything. In *XXIII BOARD GAME STUDIES COLLOQUIUM- The Evolutions of Board Games*, Paris, France, April 2021.

[36] Cameron Browne. *A Class Grammar for General Games*, page 167–182. Springer International Publishing, 2016.

[37] Cameron Browne, Dennis Soemers, Eric Piette, Matthew Stephenson, and Walter Crist III. *Ludii Language Reference.* December 2020.

[38] Cameron Browne, Éric Piette, Matthew Stephenson, and Dennis J. N. J. Soemers. General board geometry, 2021.

[39] Éric Piette, Matthew Stephenson, Dennis J.N.J. Soemers, and Cameron Browne. An empirical evaluation of two general game systems: Ludii and rbg. In *2019 IEEE Conference on Games (CoG)*, pages 1–4, 2019.

[40] Graham Todd, Alexander Padula, Matthew Stephenson, Éric Piette, Dennis JNJ Soemers, and Julian Togelius. Gavel: Generating games via evolution and language models. *arXiv preprint arXiv:2407.09388*, 2024.

[41] Cameron Browne, Eric Pierre, Walter Crist, and Matthew Stephenson. The ludii games database: A resource for computational and cultural research on traditional board games. *Digital Humanities Quaterly.*

[42] Cédric Piette, Éric Piette, Matthew Stephenson, Dennis J.N.J. Soemers, and Cameron Browne. Ludii and xcsp: Playing and solving logic puzzles. In *2019 IEEE Conference on Games (CoG)*, pages 1–4, 2019.

[43] Cameron Browne, Éric Piette, Matthew Stephenson, and Dennis J. N. J. Soemers. *Ludii General Game System for Modeling, Analyzing, and Designing Board Games*, page 1–15. Springer International Publishing, 2023.

[44] Pierre Accou. *Model and play logic puzzles within Ludii.* PhD thesis, UCL - Ecole polytechnique de Louvain, 2024.

[45] Éric Piette, Matthew Stephenson, Dennis J.N.J. Soemers, and Cameron Browne. General board game concepts. In *2021 IEEE Conference on Games (CoG)*, pages 01–08, 2021.

[46] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[47] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM*, 55(3):106–113, March 2012.

[48] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[49] Maciej Świechowski, HyunSoo Park, Jacek Mańdziuk, and Kyung-Joong Kim. Recent advances in general game playing. *The Scientific World Journal*, 2015(1), January 2015.

[50] Dennis J. N. J. Soemers, Éric Piette, Matthew Stephenson, and Cameron Browne. Learning policies from self-play with policy gradients and mcts value estimates. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[51] Dennis J. N. J. Soemers, Eric Piette, Matthew Stephenson, and Cameron Browne. Manipulating the distributions of experience used for self-play learning in expert iteration. In *2020 IEEE Conference on Games (CoG)*, pages 245–252, 2020.

[52] Jean Méhat and Tristan Cazenave. Combining uct and nested monte carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

[53] Tristan Cazenave. Nested monte-carlo search. pages 456–461, 01 2009.

[54] Francesca Rossi, Peter van Beek, and Toby Walsh. Chapter 4 constraint programming. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 181–211. Elsevier, 2008.

[55] Diogo M. Costa. Computational complexity of games and puzzles. *CoRR*, abs/1807.04724, 2018.

[56] Mehmet C, Tansel Uras, and Esra Erdem. Comparing asp and cp on four grid puzzles. 589, 01 2009.

[57] Barry O'Sullivan and John Horan. Generating and solving logic puzzles through constraint satisfaction. volume 2, pages 1974–1975, 01 2007.

[58] Cameron Browne. Deductive search for logic puzzles. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, 2013.

[59] Olivier Roussel and Christophe Lecoutre. XML representation of constraint networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.

[60] Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, Cédric Piette, and Olivier Roussel. Xcsp3 and its ecosystem. *Constraints*, 25(1–2):47–69, February 2020.

[61] Gilles Audemard, Christophe Lecoutre, and Emmanuel Lonca. Proceedings of the 2023 xcsp3 competition, 2023.

[62] Mats Carlsson and Nicolas Beldiceanu. From constraints to finite automata to filtering algorithms. In David Schmidt, editor, *Programming Languages and Systems*, pages 94–108, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[63] Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. Abscon 2005. *Proceedings of CPAI'05*, 2:67–72, 2005.

[64] Christophe Lecoutre. Ace, a generic constraint solver, 2024.

[65] Charles Prud'homme and Jean-Guillaume Fages. Choco-solver. *Journal of Open Source Software*, 7(78):4708, 2022.

[66] Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint solving and planning with Picat*, volume 11. Springer, 2015.

[67] Guillaume Derval and Damien Ernst. Symbolism for modelling, reformulations, and parallelism: Maxicp-modelling. 2023.

[68] L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021.

[69] Michael Fitzgerald. *Introducing regular expressions.* " O'Reilly Media, Inc.", 2012.

[70] Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[71] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by zdds. *Algorithms*, 5(2):176–213, 2012.

[72] Shi-JimYen, Shih-YuanChiu, Cheng-WeiChou, and Jr-ChangChen. Masyu solver. In *Game Programming Workshop 2010 Paper Collection*, volume 2010, pages 82–85, nov 2010.

[73] Md. Ahsan Ayub, Kazi A Kalpoma, Humaira Tasnim Proma, Syed Mehrab Kabir, and Rakib Ibna Hamid Chowdhury. Exhaustive study of essential constraint satisfaction problem techniques based on n-queens problem. In *2017 20th International Conference of Computer and Information Technology (ICCIT)*, pages 1–6, 2017.

[74] Black and white Japanese crosswords — nonograms.org. `https://www.nonograms.org/nonograms`. [Accessed 04-01-2025].

[75] Yngvi Björnsson, Sigurður Helgason, and Aðalsteinn Pálsson. Searching for explainable solutions in sudoku. In *2021 IEEE Conference on Games (CoG)*, volume 1, pages 01–08, 2021.

[76] Nelishia Pillay. Finding solutions to sudoku puzzles using human intuitive heuristics. *SACJ*, 49:25–34, 09 2012.

[77] Techniques de sudoku pour devenir un vrai pro ! » Sudoku Megastar — sudoku.megastar.fr. `https://sudoku.megastar.fr/accueil/techniques-de-sudoku/`. [Accessed 12-12-2024].

[78] Joan Espasa Arxer, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, Matthew J. McIlree, and Alice M. Lynch. Towards generic explanations for pen and paper puzzles with muses. In Kyle Martin, Nirmalie Wiratunga, and Anjana Wijekoon, editors, *Proceedings of the SICSA eXplainable Artifical Intelligence Workshop 2021*, CEUR Workshop Proceedings, pages 56–63, July 2021. This research was supported by the Royal Society URF80015 . ; SICSA eXplainable Artifical Intelligence Workshop, SICSA XAI 2021 ; Conference date: 01-06-2021.