



"Model and Play Logic Puzzles within Ludii"

Accou, Pierre

ABSTRACT

Deduction puzzles have become increasingly popular in recent years, and are now an integral part of everyone's daily lives. However, there is still no system capable of representing them all effectively. Ludii offers a solution with its ludemic approach. Ludii is a framework for the modelling and playability of games of various categories. This thesis focuses on the modelling of various deduction puzzles belonging to different categories using this method. The aim is to optimise these modelling methods so that they can be used later by different solvers. Various tests and experiments will be carried out to demonstrate the effectiveness of these models.

CITE THIS VERSION

Accou, Pierre. *Model and Play Logic Puzzles within Ludii*. Ecole polytechnique de Louvain, Université catholique de Louvain, 2024. Prom. : Piette, Eric. <http://hdl.handle.net/2078.1/thesis:45644>

Le répertoire DIAL.mem est destiné à l'archivage et à la diffusion des mémoires rédigés par les étudiants de l'UCLouvain. Toute utilisation de ce document à des fins lucratives ou commerciales est strictement interdite. L'utilisateur s'engage à respecter les droits d'auteur liés à ce document, notamment le droit à l'intégrité de l'oeuvre et le droit à la paternité. La politique complète de droit d'auteur est disponible sur la page [Copyright policy](#)

DIAL.mem is the institutional repository for the Master theses of the UCLouvain. Usage of this document for profit or commercial purposes is strictly prohibited. User agrees to respect copyright, in particular text integrity and credit to the author. Full content of copyright policy is available at [Copyright policy](#)

École polytechnique de Louvain

Model and Play Logic Puzzles within Ludii

Author: **Pierre Accou**

Supervisor: **Éric PIETTE**

Readers: **Kim MENS, Achille MORENVILLE, Julien LIENARD**

Academic year 2023–2024

Master [60] in Computer Science

CONTENTS

Acknowledgments	vii
1 Introduction	1
2 Background	3
2.1 The puzzles	3
2.2 Modelling and solving deduction puzzles.	5
2.3 Ludii & the ludemic approach	6
2.3.1 Origins of Ludii	9
2.3.2 The ludemic approach	11
2.3.3 Ludii Data Structure	14
2.3.4 Ludii & Deduction Puzzle	15
3 Deduction puzzle in ludii	17
3.1 The puzzles	17
3.1.1 Sudoku and his variants	17
3.1.2 Kakuro	19
3.1.3 Magic Square & Magic Hexagon	19
3.1.4 N Queens	19
3.1.5 Fill A Pix	20
3.1.6 Futoshiki	21
3.1.7 Slitherlink	21
3.1.8 Latin Square	22
3.1.9 Squaro	22
3.1.10 Takuzu	22
3.2 Graphical representations	23
3.3 Ludemes and puzzle mechanisms.	24
3.3.1 IsSolved	24
3.3.2 Satisfy	24
3.3.3 IsCount	24
3.3.4 IsSum	25
3.3.5 AllDifferent	25
4 New Rules and New Deduction Puzzle	27
4.1 Ludemes updated	27
4.1.1 IsSum	27
4.1.2 IsCount	27
4.1.3 SuperLudeme Is	29

4.2	New Ludemes	29
4.2.1	IsTilesComplete	30
4.2.2	IsCountEmpty	30
4.2.3	IsValidDirection	31
4.2.4	IsMatch	31
4.2.5	IsCrossed.	32
4.2.6	IsDistinct.	32
4.2.7	IsConnex.	32
4.2.8	Super Ludeme At.	33
4.2.9	AtMost & AtLeast	33
4.2.10	AllHintDifferent	33
4.3	The new puzzles	33
4.3.1	The Sudoku Variants	34
4.3.2	Other puzzles	39
4.3.3	Akari.	40
4.3.4	Hitori	41
4.3.5	Ripple Effect	41
4.3.6	Hashiwokakero	42
4.3.7	Nonogram	43
4.3.8	Color Nonogram	44
4.3.9	Hexagonal Nonogram	45
4.3.10	Masyu	45
4.3.11	Kurodoko	46
4.3.12	Usowan	47
4.3.13	Big Tour	48
4.3.14	Buraitoraito	48
4.3.15	Tilepaint	49
4.4	Graphic additions	49
4.4.1	what already exists.	49
4.4.2	Changes to the system	51
4.4.3	New styles	52
4.5	Ludemeplex	53
5	Testing and Experimentation	55
5.1	The tests	55
5.1.1	Integrity Test.	55
5.1.2	The JUnitTests	56
5.2	Experiments	56
5.2.1	The size of the challenge	56
5.2.2	The number of movements in 40 seconds.	57
5.2.3	Analysis and similarity.	60
5.2.4	The number of tokens per representation	61

6	Future work in connection with this thesis	63
6.1	General Game Playing and Constraint Programming to Solve any Logic Puzzles: Tom Doumont's Thesis	63
6.2	Possible future work	63
6.2.1	Addition of many other games and new rules	63
6.2.2	Creating AI to solve puzzles	66
6.2.3	Generating new instances	66
7	Conclusion	67
	Bibliography	69
	Appendix	73
7.1	Appendix A : Result experiement size of challenge	73
7.2	Appendix B : Result experiment the number of token	77
7.3	Appendix C : All the deduction puzzles in Ludii	78

ACKNOWLEDGMENTS

At the end of our dissertation, we would like to thank all the people who helped us in any way with our research.

First of all, to Professor Eric Piette, our promoter, for his invaluable advice and all the time he gave us to complete our work. His availability and help were more than essential to the completion of this thesis.

We would also like to thank Professor Kim Mens, Julien Liénard and Achille Morenville for agreeing to sit on our jury to assess this thesis.

We would like to express our gratitude to Achille Morenville for helping us set up our integrity tests on our personal GitHub.

We would also like to thank Joshua Deraeck for his help in proofreading this thesis report.

We would like to express our gratitude to Matthew Stephenson for his help in the graphics part of the system.

Finally, we would like to thank our friends and family for their support over the past year.

The DeepL tool was used to help us formulate certain sentences. The ChatGPT tool was used to help us with the layout of the report and the implementation of regular expressions.

1

INTRODUCTION

Puzzles have become quite popular and can be found in general magazines, specialized puzzle magazines, and even mobile applications. Companies like Nikoli edit and publish new puzzles and puzzle instances. These puzzles have become so popular that some people have specialised in certain puzzles (like Wayne Gould¹ with Sudoku). As shown in [15], Nikoli is the world's leading producer of deduction puzzles on paper. It is based in Tokyo, Japan. The popularity of these puzzles means that there is a need for something new to keep them fresh. To maintain the enthusiasm and popularity of puzzles, we need to innovate. There are so many of these puzzles that it is necessary to have a single system capable of representing and modelling them all and making them playable. This system could then try to solve them. A solving system could help to provide this novelty. To use these puzzles in a solving system, the different puzzles need to be represented and modelled.

[13] shows a study on solving the Sudoku puzzle using Constraint Programming (CP) approaches. This study shows results that are influenced by the selection of variables and their values. [28] uses the same approach but with another puzzle: Kakuro. In this study, different models were implemented and tested to optimise the search. Unfortunately, the proposed models are too specific to this puzzle and are not general. Currently, no system can model and solve all puzzles using constraint programming. And this is also the case for other techniques, not only constraint programming.

The purpose of this thesis is to use a General Game system[26] (Ludii[24] in this case) to model the mechanics of all deduction puzzles. Using Ludii would enable four things:

1. To have a common graphical interface for all the deduction puzzles.
2. Make it easy to design new puzzles, so that designers of all levels can test out their new ideas.
3. To allow players to solve the puzzles.
4. Integrate and combine CSP solvers with Ludii to solve puzzles.

¹<http://www.waynegouldpuzzles.com/sudoku/>

1

Ludii uses a ludemic approach[31]. These are ludemes that define each element of the puzzle. This approach allows that different ludemes can be reused in different ludemic representations. This approach makes it easy to add new mechanics. In addition, Ludii offers the possibility of converting game instances to XCSP[6] as mentioned in [20]. This conversion is possible because Ludii uses ludemes. These ludemes will be created in such a way as to be similar to the main global constraints, making them fairly easy to convert. This thesis will deal with the modelling of the puzzle and its resolution by a human. A second thesis is in progress: General Game Playing and Constraint Programming to Solve any Logic Puzzles, which deals with puzzle solving using the XCSP concept and several solvers. It is for this reason that this thesis seeks to preserve a description for the puzzles that will allow them to be solved.

The first chapter of this thesis will present the different concepts and tools that have been used. Next, a description of what the system already contains for deduction puzzles will be presented. The third section will discuss all our contributions to the system in terms of puzzles (new puzzles, new mechanics, etc.). A chapter on testing and experimentation will highlight the effectiveness of our modelling and areas for optimisation. We will also discuss some ideas for the future that this thesis raises.

2

BACKGROUND

2.1 THE PUZZLES

This thesis focuses on the modelling and representation of deduction puzzles. To introduce this topic, it is therefore necessary to define precisely what a puzzle is. [15] define it in their taxonomy of logic puzzles as "a problem with defined steps that aims to reach one or more predefined solutions such that the challenge contains all the information necessary to reach said solution". Unlike a game, the essence of a puzzle lies in the fact that it has one or more solutions and that the player has all the information needed to solve it. There is no random factor involved.

In this thesis, we will follow this taxonomy [15] (fig:2.1) to categorise the various puzzles. In this taxonomy, puzzles are classified according to their objective and the process used to reach the solution. It should be stressed, however, that we will be focusing here solely on logic puzzles. There are also mechanical (dexterity) and procedural puzzles, but these are not the subject of this work.

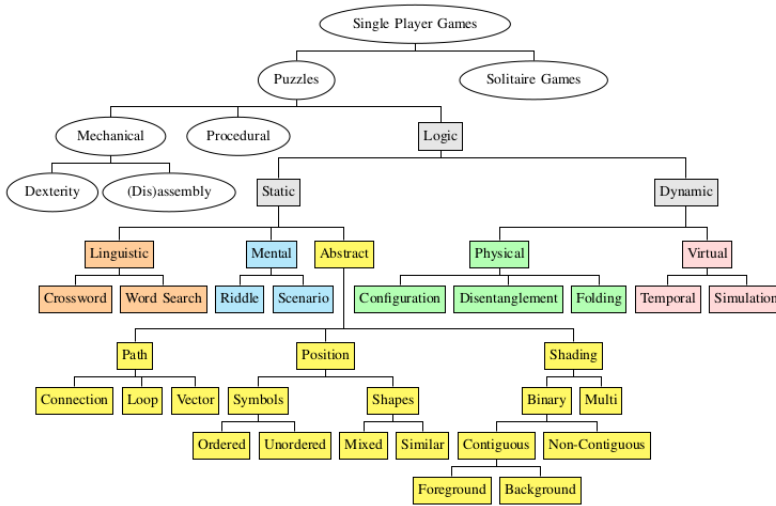


Figure 2.1: Taxonomy of Lianne V. Hufkens and Cameron Browne

Within these logic puzzles, there are deduction puzzles and planning puzzles. The difference between these two types of puzzles lies in the way they are solved. Planning puzzles require a strategy to anticipate the moves needed to solve the puzzle. This category includes games such as Tower of Hanoi¹, where it is required to move discs of increasing diameter from one tower to another without placing a disc of larger diameter on top of a smaller one. The key to these puzzles is the order of the movements. That is why they need to be planned in order to arrive at a solution.

The deduction puzzles, on the other hand, are solved on the basis of the various hints made available to us. These are single-player puzzles with simple rules that can be solved by deduction. Using information provided by the board, hints, etc., it is possible to reach a solution. This category includes games such as Sudoku², where you have to fill in a grid with numbers based on those already in the grid.

In the taxonomy, these deduction puzzles are placed in the "abstract" category.

¹https://en.wikipedia.org/wiki/Tower_of_Hanoi

²<https://www.nikoli.co.jp/en/puzzles/sudoku/>

There are 3 types of puzzles in this category:

- Path puzzles: A path, connections or arrows are drawn to indicate how to solve the puzzle (fig: 2.2)

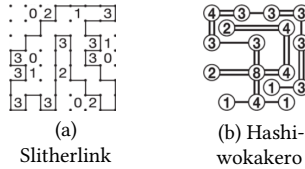


Figure 2.2: Path puzzles

- Position puzzles: To find the solution, you need to use symbols (numbers, letters, etc.) or represent different shapes (fig: 2.3).

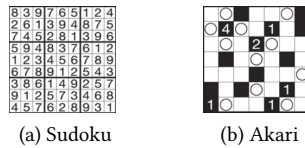


Figure 2.3: Symbol puzzles

- Shading puzzles: In these puzzles, you have to colour in certain elements to find the solution (fig: 2.4)

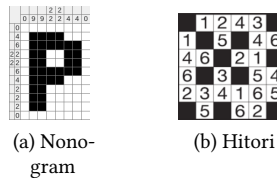


Figure 2.4: Shading puzzles

2.2 MODELLING AND SOLVING DEDUCTION PUZZLES

Various systems have attempted to model and solve certain puzzles. As shown in [17], a particular solver has been implemented for single-agent stochastic puzzles. These games are a far cry from deduction puzzles, since they use a notion of randomness. Another study [7] focuses on deductive puzzles. In this one, deductive search will be used to try and solve Sudoku and Slitherlink. Constraints will be used to train a model. These solving systems

could work depending on the puzzle it is trying to solve. To be able to solve them, they first need to be modelled in order to understand them, but also to be able to represent the puzzle with constraints. So we need to find a system that can represent all the puzzles, play with them and solve them using a solver.

2.3 LUDII & THE LUDEMIC APPROACH


One way of modelling these puzzles is through the use of General Game Playing (GGP). The aim of GGP is to develop agents capable of playing a wide variety of games [26]. In 2005, a General Game system (GGP-BASE) became a standard in research with the Game Description Languages (GDL) [18]. As explained in [24], with this GDL, you have to start from scratch every time. You have to rewrite everything in a language based on first-order logic, which is "Time Consuming". This problem applies to all types of games, including puzzles. Because of this, applications outside the field of AI are limited. Finally, game descriptions are difficult to understand and do not always represent concepts in the way that humans perceive them. Two solutions to this problem have emerged: Regular Boardgames language[16] and Ludii [24]. The first solution will make it possible to model certain complex games such as Amazons or Go. However, this system has its limitations and does not allow all the most complex games to be modelled or made playable, given the time it takes to use them. The second solution is Ludii.

Ludii, unlike other GGS, uses the Ludemic approach[24]. This approach is composed of ludemes. As shown in [24], a ludeme can be seen as a conceptual unit of information related to the game. It represents its equipment and its rules. The equipment is represented using different geometric concepts. As shown in [9], the topology and geometry of the system has also been generalised through the use of ludemes. The reasons for using this approach are as follows:

- **Simplicity:** It is easy to create and modify descriptions for our games because they use far fewer tokens. Representations using other approaches such as GDL require many more characters to represent the same game. Tokens are the different words used to represent a game.
- **Clarity:** One of the aims of the ludemic game description is to make it understandable to someone unfamiliar with the game. This is done by choosing the names of the ludemes. The ludemes have names that directly represent their behaviour and that use terms close to what is commonly used in jargon.
- **Generality:** The general aspect of a ludeme allows it to be reused in multiple games. The general aspect means that it adapts to different situations (depending on the equipment it has, for example).
- **Extensibility:** It is easy to create new ludeme. The new ludemes are coded in new classes. These classes are added directly to the system and the new elements are therefore directly available in the grammar.
- **Scalable:** Ludemes are easily adaptable and avoid the need to recreate new elements each time.

- Effectiveness: As demonstrated in the publication [25], the ludemic approach, compared with the Regular Boardgames language[16] and compared to GGP-BASE[26], will offer the ability to make many more moves per second for the vast majority of games. These observations are possible thanks to various optimisation processors for calculating legal costs and simulating games (playouts) as described in [30].

To illustrate these different advantages, the first figure shows the description of Tic-Tac-Toe³ using the GDL principle (fig: 2.6). The second shows the Tic-Tac-Toe using the ludemic representation (fig: 2.5).



Ludeme

```

1  (game "Tic-Tac-Toe"
2    (players 2)
3    (equipment {
4      (board (square 3))
5      (piece "Disc" P1)
6      (piece "Cross" P2)
7    })
8    (rules
9      (play (move Add (to (sites Empty))))
10     (end (if (is Line 3)(result Mover Win)))
11   )
12 )

```

Figure 2.5: Tic-Tac-Toe with ludemic representation

³<https://en.wikipedia.org/wiki/Tic-tac-toe>

Example 2.1: GDL of Tic-Tac-Toe

```

1 (role white) (role black)
2 (init (cell 1 1 b)) (init (cell 1 2 b)) (init (cell 1 3 b))
3 (init (cell 2 1 b)) (init (cell 2 2 b)) (init (cell 2 3 b))
4 (init (cell 3 1 b)) (init (cell 3 2 b)) (init (cell 3 3 b))
5 (init (control white))
6 (<= (legal ?w (mark ?x ?y)) (true (cell ?x ?y b))(true (control
   ?w)))
7 (<= (legal white noop) (true (control black)))
8 (<= (legal black noop) (true (control white)))
9 (<= (next (cell ?m ?n x)) (does white (mark ?m ?n)) (true (cell ?m
   ?n b)))
10 (<= (next (cell ?m ?n o)) (does black (mark ?m ?n)) (true (cell ?m
   ?n b)))
11 (<= (next (cell ?m ?n ?w)) (true (cell ?m ?n ?w)) (distinct ?w b))
12 (<= (next (cell ?m ?n b)) (does ?w (mark ?j ?k))
13     (true (cell ?m ?n b)) (or (distinct ?m ?j)
14     (distinct ?n ?k)))
15 (<= (next (control white)) (true (control black)))
16 (<= (next (control black)) (true (control white)))
17 (<= (row ?m ?x) (true (cell ?m 1 ?x))
18     (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
19 (<= (column ?n ?x) (true (cell 1 ?n ?x))
20     (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
21 (<= (diagonal ?x) (true (cell 1 1 ?x))
22     (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
23 (<= (diagonal ?x) (true (cell 1 3 ?x))
24     (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
25 (<= (line ?x) (row ?m ?x))
26 (<= (line ?x) (column ?m ?x))
27 (<= (line ?x) (diagonal ?x))
28 (<= open (true (cell ?m ?n b)) (<= (goal white 100) (line x))
29 (<= (goal white 50) (not open) (not (line x)) (not (line o)))
30 (<= (goal white 0) open (not (line x)))
31 (<= (goal black 100) (line o))
32 (<= (goal black 50) (not open) (not (line x)) (not (line o)))
33 (<= (goal black 0) open (not (line o)))
34 (<= terminal (line x))
35 (<= terminal (line o))
36 (<= terminal (not open))

```

Figure 2.6: Tic-Tac-Toe with GDL representation

The GDL representation of Tic-Tac-Toe is three times bigger than the one in Ludii. As well as being much longer, it is much less clear. It is understandable that the game is two-player and that nine cells are initialised. What is done in nine initialisations is done in one line with the ludemic approach. The authorised moves are difficult to understand with the second approach (fig: 2.6), as are the victory conditions. In the first approach (fig: 2.5), it is understandable that the player places a piece on an empty space and that the player wins when a line of three is made.

2.3.1 ORIGINS OF LUDII

The origins of Ludii start in 2006, when in Brisbane, Australia, Cameron Browne [12] designed Ludi. It is the previous project which led to the development of the General Game System Ludii. It is the first system that used the ludemic approach. More than two hundred ludemes were implemented in this version. This approach has led to the development of a number of board games (e.g. Yavalath⁴). In fact, the system is the first to model a new game using only computers. No human action was required. The Ludi approach was not sufficiently general, and the language was not easily extensible. Indeed, due to an implementation in C++, when a new game was added, it was generally necessary to update the grammar and the code. With this method of representation, it was mainly necessary to modify the compiler rather than add new mechanics.

As a result, a new type of grammar has been developed to overcome this problem. This is how Class Grammar[8] was developed. As shown in [24], Ludii presented the use of this approach, which maps a keyword in the representation to the associated Java code using Java Reflection. The Class Grammar facilitated the modelling of the game by recovering the different constructors in the code linked to the different keywords. All this to ensure a one-to-one correspondence between the source code and the grammar. As described in [8], this strategy leaves developers free to make their own implementation choices. In this way, the user will only see the simplified version of the constructor in the grammar and not its full implementation.

Launched in Maastricht in 2018, the Digital Ludeme Project⁵ aims to study the evolution of traditional strategy games[11] to model them.

This project covered multiples game-related topics such as:

- Improve understanding of games using AI techniques.
- Trace the historical development of certain games.
- Explore the role of games in different cultures.

Games dating back thousands of years are often incomplete, and information about the rules and equipment is often missing. Ludii was created with this in mind. This platform uses AI techniques to reconstruct these different games. [14] and [23] are studies of older games. They were used to identify missing rules and to determine, for example, the size of

⁴<http://cambolbro.com/games/yavalath/>

⁵<http://ludeme.eu/>

the board by carrying out various tests with what they had available. This system has been used to model and understand more than 1,000 games. All these games are represented in a database. In addition to the study, Ludii offers the possibility of playing these games and adding new ones. This was made public for the first time in July 2020^{6,7}. From now on, all the work carried out in this field will be carried out by an international group of researchers called GameTable, which has been made possible by COST⁸ [22].

Since then, Ludii has gone from strength to strength, with 1,299 games of various types (board games, dominoes, etc.) now in its directory (fig2.7).

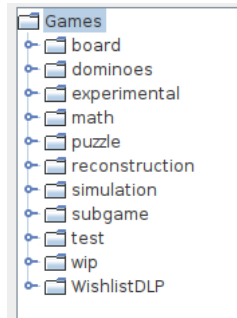


Figure 2.7: Type of game available in Ludii

In its latest public version 1.3.12, published on 01-08-2023, the Ludii system included 1,299 games. This platform has made it possible to represent games with unconventional boards. Internally, the boards will be represented as graphs made up of vertices, edges and faces[9]. Thanks to this generation of graph, it is possible to generate games with a particular board by specifying the coordinates of the vertices. In addition, this process offers the option of playing on one of the three possible sites (edges, vertices, cells) (fig: 2.8).

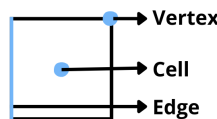


Figure 2.8: The different SiteType

Once the board has been generated, two relationships are automatically calculated for each site:

- Steps: steps to an adjacent element (if any). It is not necessary to specify a direction.
- Radials: Calculate how to get to an adjacent element with as few changes of direction as possible. All this helps with line detection or visibility tests, for example. These

⁶<https://ludii.games/>

⁷<https://github.com/Ludeme/Ludii>

⁸<https://www.cost.eu/actions/CA22145/>

calculations are carried out before the game is launched. This will save time during the game, as these calculations will not need to be repeated. The drawback is that if there are a lot of elements in the graph, these calculations can be relatively long.

2.3.2 THE LUDEMIC APPROACH

2

All games within the General Game System Ludii are designed using the Ludemic Approach. This method employs "ludemes" to systematically describe game components and functions in a hierarchical tree structure. Elements such as the board, pieces, rules, and other game-related details are articulated using this approach. It can be a piece of game equipment, such as the board or part of the board, or even the rules.

From a technical point of view, game elements are implemented in Java classes. A class corresponds to a game element. Each element in the game representation (string, integer, etc.) can be seen as a Java class. Here is the game description of the Amazon game in the Ludii Game Logic Guide[21] [29].

```

❖ Ludeme
1  (game "Amazons"
2    (players 2)
3    (equipment {
4      (board (square 10))
5      (piece "Queen" Each (move Slide (then (moveAgain))))
6      (piece "Dot" Neutral)
7    })
8    (rules
9      (start {
10       (place "Queen1" {"A4" "D1" "G1" "J4"})
11       (place "Queen2" {"A7" "D10" "G10" "J7"})
12     })
13
14     (play
15       (if (is Even (count Moves))
16         (forEach Piece)
17         (move Shoot (piece "Dot0"))
18       )
19     )
20     (end (if (no Moves Next) (result Mover Win)))
21   )
22 )

```

This game is organised into different sections (players, equipment and rules). In these sections, we will find a set of ludemes describing the section, these same words referring to a Java class present in Ludii[10]. These classes define the 'Ludeme' interface (fig: 2.9), which can be seen as the top of a hierarchy.

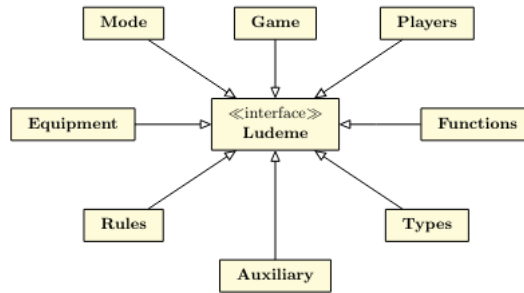


Figure 2.9: The Ludeme diagram from the Ludii Game Logic Guide

It is important to note that different ludeme types are available. For example, there are:

- RegionFunctionLudemes: used to obtain a region
- IntArrayFunctionLudemes: retrieve an array of integers
- IntFunctionLudemes: return an integer
- BooleanFunctionLudemes: return a boolean to indicate whether something is True or False.

In our case, we will mainly be using BooleanFunction ludemes to determine whether one of our conditions in a puzzle is considered to be True or False. A problem is defined as such in the code. The ludeme is defined using two main elements:

- A constructor: This is the element that will enable us to call up the ludeme. It is in this constructor that we will decide which arguments the ludeme will take.



```

1  public IsCount(
2      @Opt final SiteType      type ,
3      @Opt final RegionFunction region ,
4      @Opt final IntFunction  what ,
5      final IntFunction      result
6  )
7  {
8      this.region = region;
9      whatFn = (what == null) ? new IntConstant(1) :
        what;
10     resultFn = result;
11     this.type = type;
12 }
  
```

- An "eval" function: This is where we will find the logic behind the ludeme. In some cases, this function will return an element according to its type.



```

1  public boolean eval(Context context){
2      if (region == null)
3          return false;
4
5      final SiteType realType = (type == null) ?
        context.board().defaultSite() : type;
6
7      final ContainerState ps =
        context.state().containerStates()[0];
8      final int what = whatFn.eval(context);
9      final int result = resultFn.eval(context);
10     final int[] sites = region.eval(context).sites();
11
12     boolean assigned = true;
13     int currentCount = 0;
14
15     for (final int site : sites){
16         if (ps.isResolved(site, realType)){
17             final int whatSite = ps.what(site, realType);
18             if (whatSite == what)
19                 currentCount++;
20         }
21         else
22             assigned = false;
23     }
24
25     if ((assigned && currentCount != result) ||
        (currentCount > result))
26         return false;
27
28     return true;
29 }

```

2

The notion of hierarchy was introduced earlier. The ludemes can be seen as a tree where a higher element (a ludeme) will call upon a lower element (another ludeme). The ludeme Is is a good example. This ludeme is called a super-ludeme because it calls on other ludemes, as shown in the figure below (fig. 2.10).

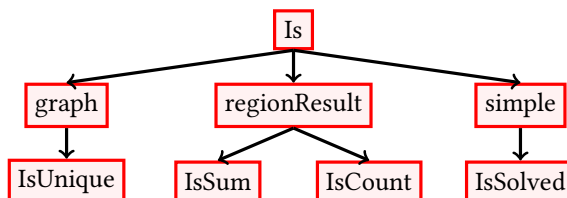


Figure 2.10: The super ludeme Is

The super ludeme has the IsCount and IsSum rules, which are in the category of rules that apply to RegionResult because we are performing an operation on a certain region. We also have the IsSolved which belongs to the simple category. The difference between these two categories is the arguments that the rules require. IsUnique in the graph category will check elements on the graph.

2.3.3 LUDII DATA STRUCTURE

In this subsection, we focus on the Ludii representation of puzzle[21]. The system implemented a Java object "context" containing:

- A static object encapsulating all information compiled from the ludemic game description.
- A game state representation which models the current positions of the pieces/elements on the board. This object evolves after applying any move to the current state. Commonly speaking, this representation is called a forward model.
- The trial which is the player's sequence of play from the beginning.

In addition to the "context" object, the different state of games will be represented as state object. Within this thesis, the representation of the start of the game will be called the initial state and the representation of the solution to the puzzle will be called the final state. These states are stored in state containers. Ludii has different types of state, depending on the type of game.

The container will be represented as a graph with cells, vertices and edges. Thanks to this container, it is possible to find locations made up of the container in question, the type of site, the site index and its level. These levels are useful for stacking games (which we will not use in this thesis). As with states, there are different types of container states for storing game elements in a more optimised way so that information can be accessed more easily. The container state is made up of a set of data vectors. These vectors are represented in the form of Bitsets, called ChunkSets in the system. This representation is used to compress the information for memory.

Deduction puzzles will be represented slightly differently. Indeed, the states of these puzzles are represented similarly as Constraint Satisfied Problems [2] do. These states will be composed of two elements:

- A set of variables, each of which has a predefined domain.
- A set of constraints linking each of these variables.

Here is the NQueens puzzle:



Ludeme

```

1 (game "N Queens"
2   (players 1)
3   (equipment {
4     (board (square 8) (values Cell (range 0 1)))
5     (regions {AllDirections})
6   })
7
8   (rules
9     (play
10      (satisfy {
11        (is Count (sites Board) of:1 <Size>)
12        (all Different except:0)
13      })
14    )
15    (end (if (is Solved) (result P1 Win)))
16  )
17 )

```

2

This puzzle is organised in a square board of size eight by eight and therefore a total of sixty-four cells which can have the value one or zero. The domain of each variable (cells on the board) will therefore be {0,1}. Each variable will be represented in the container state (ContainerDeductionPuzzleState). The BitSet associated with the variable will indicate the available values in the domain of this variable.

Some methods are associated with this type of container state to modify BitSets. These include :

- bit(var, value): indicates whether a value is available in the domain of a variable.
- set(var, value): Sets the domain with the associated value.
- toggle(var, value): changes the value bit in the variable
- reset(var): resets all bits in the Bitset to True so that all values are available
- isResolved(var): returns true if the variable's domain is fixed
- what(var): returns the value set in the variable's domain.

2.3.4 LUDII & DEDUCTION PUZZLE

In the current version of Ludii, nineteen deduction puzzles are available, including Sudoku and some of its variants. These include Sudoku, Anti-Knight Sudoku, Tridoku, Killer Sudoku, Squaro, Slitherlink, Kakuro, Magic Square, Magic Hexagon, ...

What all these puzzles have in common is that they involve filling in a grid. Using the different elements, we can create a set of versions of the puzzle. For example, it is possible to create a multitude of Sudoku with just the basic grid. If we take chess for comparison, the starting positions of the different pieces will always be the same.

3

3

DEDUCTION PUZZLE IN LUDII

As mentioned above, some deduction puzzles are already parts of the Ludii Library. In this section, we will present the puzzles that have already been implemented, as well as the various rules and ludemes that will be useful to us.

3.1 THE PUZZLES

3.1.1 SUDOKU AND HIS VARIANTS

In this subsection, we focus on Sudoku¹, the most popular puzzle type worldwide. Sudoku (fig: 3.1) consists of a board with eighty-one cells divided into a nine by nine square. You have to fill in the different cells according to three rules:

1. The same number cannot appear twice in each row.
2. In each column, we cannot have the same number twice.
3. On each sub-grid of nine squares, we cannot have the same number two times.

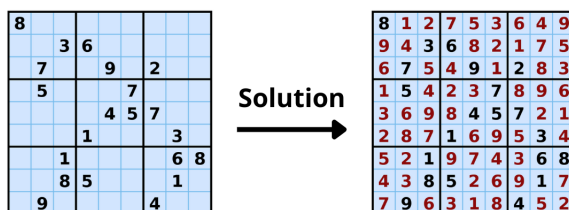


Figure 3.1: An Empty Sudoku and its solution

Sudoku exists in several different variants (fig: 3.2). There is Sudoku X, for example, which adds two new regions to solve - the two diagonals of the board. Ludii also has the Killer

¹<https://www.nikoli.co.jp/en/puzzles/sudoku/>

Sudoku, which is a Sudoku with lots of mini regions in addition to the classic ones. These new regions are composed of a hint, and the sum of the contents of these regions must be equal to this hint. Here is the list of sudoku and their variants already available:

- Sudoku
- Anti-Knight Sudoku
- Killer Sudoku
- Samurai Sudoku
- Sudoku Mine
- Sudoku X
- Tridoku
- Hoshi
- Sujiken

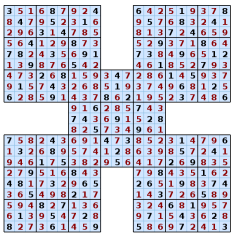
3



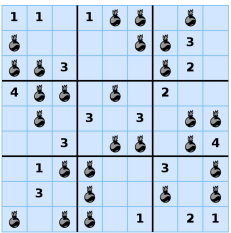
(a) Antiknight
Sudoku



(b) Killer Sudoku



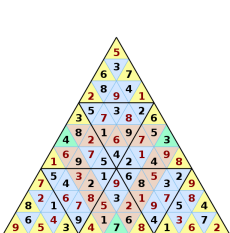
(c) Samurai
Sudoku



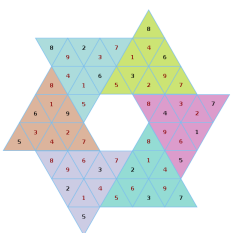
(d) Sudoku Mine



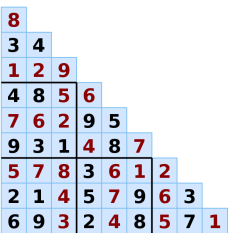
(e) Sudoku X



(f) Tridoku



(g) Hoshi



(h) Sujiken

Figure 3.2: Sudoku variants

3.1.2 KAKURO

Kakuro² (fig: 3.3) is another very popular game. It consists of a rectangle with X rows and Y columns. In this board, the puzzle has several regions, each with an associated hint that will be located either to the North or to the West of the region. In this region, we have to place numbers in order to add up the sum that corresponds to the hint. You can not have the same number twice in the same region. Compared to Sudoku, there are several sizes.

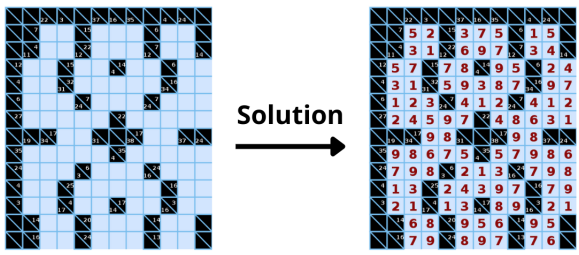


Figure 3.3: An Empty Kakuro and its solution

3.1.3 MAGIC SQUARE & MAGIC HEXAGON

The next puzzle is the Magic Square³ (fig: 3.4). This puzzle is a square into which we have to place numbers. In this square, all the numbers must be different. To solve the puzzle, the sum of each row, each column and the two diagonals must be the same. This value is predefined according to the size of the square.

Magic Hexagon (fig: 3.5) is a variant of Magic Square. This time, the board is no longer a square but a hexagon. This time, the sum must be made on the rows and diagonals.

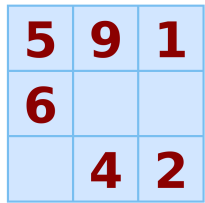


Figure 3.4: MagicSquare in Ludii

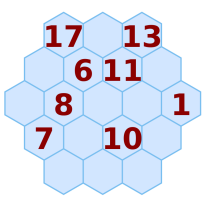


Figure 3.5: Magic Hexagon in Ludii

3.1.4 N QUEENS

In this puzzle⁴ (fig: 3.6), it is not numbers but queens that need to be placed on an n by n board. The n queens must be placed on the board in such a way that no other is on the same row, column or diagonal. It is an interesting puzzle because, as this implementation

²<https://www.nikoli.co.jp/en/puzzles/kakuro/>

³https://en.wikipedia.org/wiki/Magic_square

⁴https://en.wikipedia.org/wiki/Eight_queens_puzzle

idea⁵ shows, it is a problem that has already been studied and was therefore important to model.

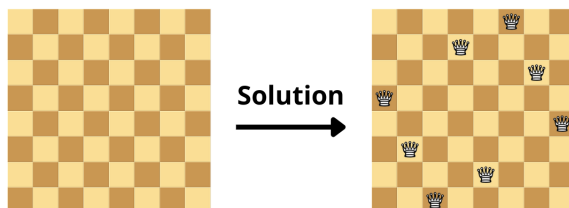


Figure 3.6: An Empty N Queens and its solution

The ludemic game description of the puzzle is described as follows:

Ludeme

```

1 (game "N Queens"
2   (players 1)
3   (equipment {
4     (board (square 8) (values Cell (range 0 1)))
5     (regions {AllDirections})
6   })
7
8   (rules
9     (play
10      (satisfy {
11        (is Count (sites Board) of:1 <Size>)
12        (all Different except:0)
13      })
14    )
15    (end (if (is Solved) (result P1 Win)))
16  )
17 )

```

3.1.5 FILL A PIX

Fill A Pix⁶ (fig: 3.7) consists of a square board made up of several hints. Each hint represents a sub-board of nine squares where the hint is the centre of that region. It indicates how many squares need to be coloured in the region. Once the puzzle has been solved, an image appears showing a drawing.

⁵https://medium.com/@carlosgonzalez_39141/using-ai-to-solve-the-n-queens-problem-2a5a9cc5c84c

⁶<https://www.cross-plus-a.com/puzzles.htm#FillAPix>

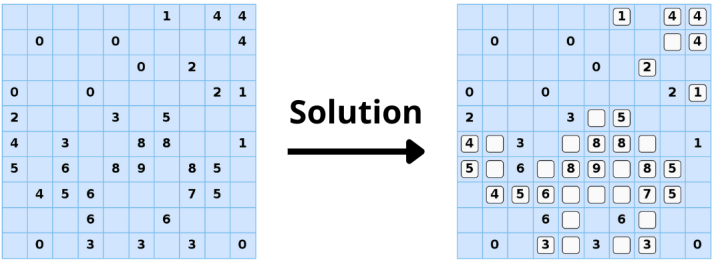


Figure 3.7: An Empty Fill A Pix and its solution

3.1.6 FUTOSHIKI

This game⁷ (fig: 3.8) is made up of a square board. The player needs to place numbers to complete the grid. Each row and column must contain different numbers. Symbols smaller than and greater than are also present to indicate the conditions to the player.

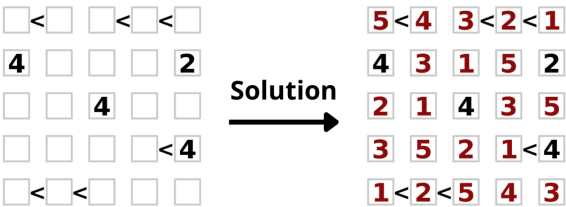


Figure 3.8: An Empty Futoshiki and its solution

3.1.7 SLITHERLINK

Unlike other puzzles, this one⁸ (fig: 3.9) won't be played on cells, but on edges. The aim is to complete a loop. This loop can not cross, and there are hints on some of the cells. This number indicates the number of edges that must be part of the loop for this cell.

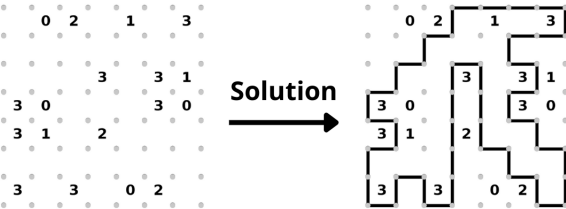


Figure 3.9: An Empty Slitherlink and its solution

⁷<https://fr.wikipedia.org/wiki/Futoshiki>
⁸<https://www.nikoli.co.jp/en/puzzles/slitherlink/>

3.1.8 LATIN SQUARE

The aim of Latin Square⁹ (fig: 3.10) is to complete the square grid of size n by n with numbers from one to n . As with many of the games already presented, the aim is to have different numbers in each row and column.

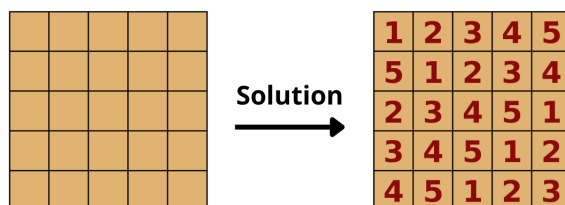


Figure 3.10: An Empty Latin Square and its solution

3.1.9 SQUARO

Squaro¹⁰ (fig: 3.11) is a deduction puzzle in which we will be playing with vertices. The aim of the game is to place black pieces on the vertices of a square board. The cells on the board contain hints that will show how many pieces are waiting on the vertices of the cell.

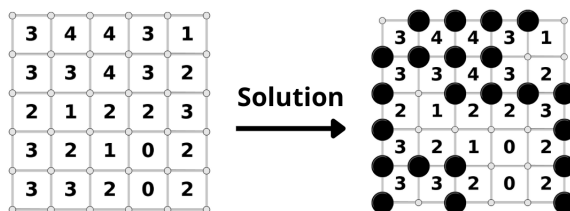


Figure 3.11: An Empty Squaro and its solution

3.1.10 TAKUZU

Takuzu¹¹ (fig: 3.12) is a binary game, where the player has to place ones or zeros on the square board. The rule is that the participant can not have more than two consecutive numbers. Numbers are already placed to guide us towards the final solution.

⁹https://en.wikipedia.org/wiki/Latin_square

¹⁰<https://www.nordinho.net/vbull/puzzles/35331-squaro.html>

¹¹<https://en.wikipedia.org/wiki/Takuzu>

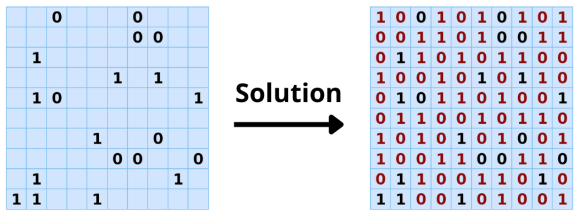


Figure 3.12: An Empty Takuzu and its solution

3.2 GRAPHICAL REPRESENTATIONS

Graphics are represented in Ludii using a library : Graphics2D¹². Each puzzle does not have its own graphic style. There is a general graphic style for all games (unless we select one). In Ludii, we can select metadata to customise the graphics of our puzzles. These metadatas offer a range of possibilities, such as renaming a piece to put a symbol in place of a number for example, or giving a certain colour to a piece. (fig: 3.13 & 3.14)

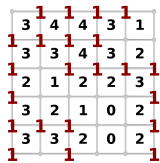


Figure 3.13: Squaro without metadata graphics

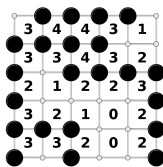


Figure 3.14: Squaro with metadata graphics

There are also a number of specific graphic styles used to make games more attractive. Take Slitherlink, for example (fig: 3.15 & 3.16), which uses the "PenAndPaper" style. The player plays on the edges, so making the cells and edges disappear is a real way of making the game more readable for the player. Sudoku also has a particular graphic style because the sub-grids are drawn automatically. This style can be reused for all puzzles that are variants of Sudoku and have a conventional board (nine by nine square).

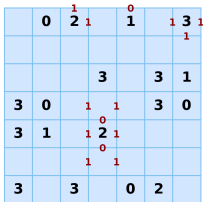


Figure 3.15: Slitherlink without Design

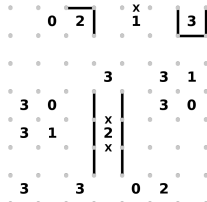


Figure 3.16: Slitherlink with Design

¹²<https://docs.oracle.com/javase/8/docs/api/java/awt/Graphics2D.html>

3.3 LUDEMES AND PUZZLE MECHANISMS

In this section, we will analyse the rules specific to the deduction puzzle already present in the public version of Ludii. In the previous chapter, we mentioned how the game is modelled using different elements. Here, we are going to focus more on the rules for solving a puzzle.

There are 3 types of rules:

3

- **Start:** These are the rules that will be applied when the game is created. Generally speaking, they involve applying certain values in certain regions in order to give the player the start of a resolution. This rule will modify our initial state with the elements it contains
- **Play:** These are the rules that will be applied while the game is running. Together, they will indicate whether the move the player wishes to make is a forbidden move, making one of the rules false. This will therefore indicate the legal moves for the current state.
- **End:** These are the rules that will be applied after each move to determine whether the puzzle is finished and solved or not. These rules will check whether all the conditions have been met, so that the puzzle can be declared solved.

3.3.1 ISOLVED

This rule will be used for all puzzles as an end rule. This rule will assign each unassigned site the lowest value in its domain. This is why, when creating a ludeme or puzzle, we will never consider the value zero to be a movement on the part of the player. It will then check all the conditions (constraints) in the play section. If all these conditions return true, then the game is solved. If not, the game continues. This rule takes no arguments.

3.3.2 SATISFY

This rule will be used for all deduction puzzles. It will test each condition (constraint). It will determine all the legal moves by combining all these conditions. It is this rule that will prevent the player from making a move that goes against the rules of the puzzle.

3.3.3 ISCOUNT

IsCount will count the number of elements present and check whether the number counted is the expected number. This can be done for a region or for the whole board. You need to specify the expected number. The possible parameters for this rule are :

- Specify which siteType we are working on (not required)
- Specify a region for which we want to check the number of elements present (not mandatory)
- Specify the value of the elements we are counting. If this is not specified, we will count all the elements that have the value one.

- As previously mentioned, you need to give the expected number of elements

If we have more elements than expected (in the event that all the values are not assigned) or if the observed value is not the right one, then we return False.

3.3.4 IsSum

IsSum will add up the different elements of a region. As with IsCount, certain parameters are possible:

- Specify the region we want to add (not required)
- Specify which siteType we are working on (not compulsory)
- You must specify the value of the expected sum.

If the result is greater than expected (without the case where all the elements are not allocated) or if the observed value is not the right one, then the rule will return False.

3.3.5 ALLDIFFERENT

AllDifferent will determine, as its name suggests, to have different values for a predefined region. Here are the main parameters, which are not mandatory:

- Specify the region we want to make unique
- Specify which siteType we are working on

The rule will return False if we want to add a value that already exists in the region. If no region is given as a parameter, the rule will check whether any regions exist. If so, it will execute on the various regions. If no regions exist, it will run on the whole board.

4

NEW RULES AND NEW DEDUCTION PUZZLE

4

This section will present the contributions made in this thesis. We will cover the ludemes we have modified, as well as all the new ludemes and puzzles we have created. New graphic styles have also been added to personalise each puzzle and model them in accordance with their original format. As a reminder, we have made these new additions and changes to represent all the deduction puzzles in Ludii.

4.1 LUDEMES UPDATED

These different games have been modified to provide new functions.:

4.1.1 IsSum

This rule still has the same function: counting and adding the value of elements in a region. This ludeme can now be applied to specific regions. These will be identified by name. In the code, the constructor already took a string argument allowing this, but the features were not coded in the ludeme's "eval" method. This change has enabled us, for example in Akari, to force a certain sum of zeros on certain cells (so that these cells are unplayable for the player).

4.1.2 IsCount

As with IsSum, IsCount has retained its original function: counting the number of elements, with the option of specifying the element to be counted. The change is in the structure of the code. The changes to this ludeme have made it possible to add the possibility of applying this condition to a specific region. This is done using the name of the region. In addition to the previous arguments, the constructor will now take a string argument to specify the name of the region.

This is the old constructor:



```

1 public IsCount(
2   @Opt final SiteType      type ,
3   @Opt final RegionFunction region ,
4   @Opt final IntFunction   what ,
5       final IntFunction    result
6 ){
7     this.region = region;
8     whatFn = (what == null) ? new IntConstant(1) : what;
9     resultFn = result;
10    this.type = type;
11 }

```

4

This is the new constructor:



```

1 public IsCount(
2   @Opt final SiteType      type ,
3   @Opt final RegionFunction region ,
4   @Opt final IntFunction   what ,
5   @Opt final String        nameRegion ,
6       final IntFunction    result
7 ){
8     this.region = region;
9     whatFn = (what == null) ? new IntConstant(1) : what;
10
11     if(region != null)
12         regionConstraint = region;
13     else
14         areaConstraint = RegionTypeStatic.Regions;
15
16     resultFn = result;
17     this.type = type;
18
19     name = (nameRegion == null) ? "" : nameRegion;
20 }

```

4.1.3 SUPERLUDEME Is

This super ludeme has been modified to include all the new rules we have added. All the rule calls are in this super ludeme. Here is the new graph associated with the super ludeme Is (fig: 4.1):

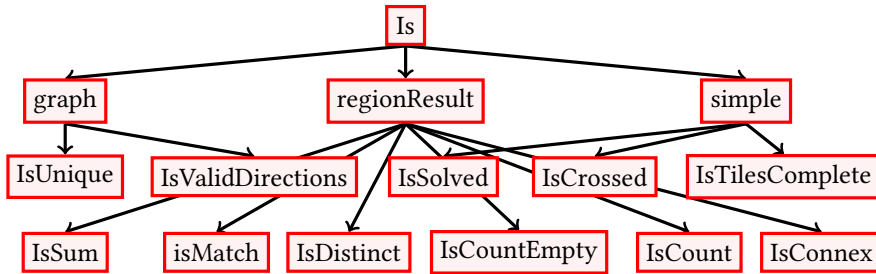


Figure 4.1: The super ludeme Is

4.2 NEW LUDEMES

All these new ludeme have been created with one aim in mind: to be as general as possible. Many of these new ludemes have evolved over the year of development so that they can be used in as many games as possible. It is important to note that all the rules will return a boolean to indicate whether or not a movement is considered prohibited. We will therefore return the boolean True when a movement is authorised, otherwise it will be the boolean False. All the rules must have been thought through with the IsSolved final condition presented in the previous chapter. As a reminder, this ludeme will assign the smallest value in its domain to all unassigned variables. All the rules have therefore had to be thought out in such a way that the game does not end if it is not finished, but also that it returns False if the move made is completely false and makes a rule impossible.

These new ludemes have also been designed to have names that are as clear as possible. As the ludemes are designed using Class Grammar[8], each ludeme is associated with a keyword. These keywords are carefully chosen for two reasons:

1. The ludemic description of the puzzle must be as clear as possible. The aim is for any reader of the description to be able to understand the puzzle (equipment and rules) even if they have no knowledge of it.
2. Ludii is a system used by many people. Among them are game designers. We therefore need a language that is clear to everyone (including people with no computer knowledge).

Sudoku is an interesting example. It is easy to understand that the board is a square, with rows, columns and sub-grids. You have to place numbers from one to nine. Some numbers are already placed on the board and each number must be different in the same region.



Ludeme

```

1 (game "Sudoku"
2   (players 1)
3   (equipment {
4     (board (square 9) (values Cell (range 1 9)))
5     (regions {Columns Rows SubGrids}))
6   })
7   (rules
8     (start (set { {1 9} {6 4} {11 8} {12 5} {16 1} {20 1}
9                 {25 6} {26 8} {30 1} {34 3} {40 4}
10                {41 5} {42 7} {46 5} {50 7} {55 7} {58 9} {60 2} {65
11                 3} {66 6} {72 8} })))
12     (play (satisfy (all Different)))
13     (end (if (is Solved) (result P1 Win))))
14 )

```

4

4.2.1 ISTILESCOMPLETE

The purpose of this new rule is to check whether a region is completely empty or completely full. This rule will run at the same time as `IsSolved` and has been created for the `Tilepaint` puzzle¹. In this puzzle, there are tiles on the board that must be filled or not according to hints. The tiles must be either empty or full, as this game suggests. This ludeme will iterate over each region with the name "Tiles" and check each site in the region. The condition will store in a counter the number of sites that have the value one. If at the end, the counter has a value other than zero or a value other than the size of the region, then the tile is not complete, and therefore the answer is wrong.

4.2.2 ISCOUNTEMPTY

`IsCountEmpty` is very similar to `IsCount` in the way it is implemented. The aim of this ludeme is to determine the number of unresolved squares in a region. This condition was created for the `Kurodoko` puzzle². In this puzzle, each hint must have the correct number of unblackened squares in an orthogonal direction. This puzzle will therefore have as its argument an orthogonal region and the index of the cell containing the hint (which is the center of the region). We will therefore iterate over the region and classify the sites present in the region to find out whether they are to the north, west, east or south of this center. Once this has been done, we will iterate on the four directions, starting from the center. The ludeme will count the number of unresolved cells. Once a resolved site is encountered (a value other than zero), we will stop iterating this part of the region and move on to the next region. At the end, the number counted will be compared to the desired number. The condition will return `True` if the number is greater than or equal to the desired number. Given that we are placing blocks to reduce the size of the region, if we have a larger counter, it means that we have a solution under construction. This is due to ludeme `Satisfy`. At

¹<https://www.nikoli.co.jp/en/puzzles/tilepaint/>

²<https://www.nikoli.co.jp/en/puzzles/kurodoko/>

present, this ludeme is adapted for square boards. It could be generalised by adapting to the shape of the board.

4.2.3 IsValidDirection

IsValidDirection is used in games where we try to make a path. It forces the player to make either a straight line or a turn, depending on the hint encountered. In this new rule, you need to find all the hints present on the vertices and the associated vertices. The condition will iterate over each edge and keep the edges linked to the vertex. The ludeme will then analyse the value of the hint: if we get a zero we want a straight line, if we get a one we want an angle. This rule will block some edges depending on where the first one was played.

4.2.4 IsMatch

IsMatch has been designed for the Nonogram and its variants³. This ludeme will determine which cells on the board can be coloured according to the associated hints. This condition works with regions. For each region, we will then determine which squares are resolved and which are not. If they aren't, we will store the index for that site. We then call up our first auxiliary function.

This function will generate all the possible cases by carrying out an exhaustive exploration of the various possible combinations for the region we are working on. In the case of a classic nonogram, the ludeme will fill in our region with either zeros to avoid selecting these boxes or ones to confirm their selection. If we use the coloured version, regions will be completed with zeros as well as other integers. Each integer corresponds to a colour. As shown in the previous chapter, logic and graphics are separate. So we reason with numbers, not colours. The ludeme are going to have to generate all the cases combinatorially. To do this, we are going to use the function recursively. As long as we have not reached the size of the region, we will first add a zero and then a one. Once we have reached the desired size, this pseudo solution will be stored in a list that we will use immediately afterwards. Finally, once all the cases have been generated, we will retrieve the list containing them all.

We will then check each pseudo-region generated. We will call a second auxiliary function to handle this. In this function, we will start by generating our pattern for our regular expression. The regular expressions⁴ are data structures that use characters to define a certain sequence. This sequence will be created according to our hints. Here is the structure that our regular expression will have: "0*1{X}0+1{Y}0*". This regular expression is based on the structure of our hints. We are going to want to have the right number of resolved cells, which are our hints. If there is several hints (and therefore several groups of resolved boxes), an unresolved box must separate these two groups. If there is one hint for a region, the regular expression will have the following structure: "0*1{X}0*". 1{X} represents the number of ones expected to follow. X being the expected number. 0+ means that we want at least one zero but that we can have more. Finally, 0* indicates that we can have a certain number of zeros without specifying a minimum (so having no zeros is possible). Once this

³<https://en.wikipedia.org/wiki/Nonogram>

⁴https://en.wikipedia.org/wiki/Regular_expression

pattern has been created, we will compare each region in our list with this pattern. If a region matches our regular expression, we will return true and stop analysing the other regions generated. If it doesn't, we will continue checking the other regions. If no region matches, this means that no solution is possible, so we will return the boolean False to indicate an impossible move.

4.2.5 IsCROSSED

This ludeme uses the edges. In this rule, IsCrossed will check whether or not two edges overlap. In this rule, we are going to retrieve all the edges and check each edge against the others. The condition are going to retrieve the vertices that determine the start and end of the edges. Then it will check the positions of the vertices to determine whether or not the edges overlap.

This ludeme was created for the Hashiwokakero⁵ and is used with the logical word not. This logical word reverses the associated rule. This ludeme will indicate whether edges overlap. In the puzzle, links must not be able to overlap.

4.2.6 IsDISTINCT

IsDistinct will be used with a region and an integer. The aim of this ludeme is to make this integer unique in this region. To do this, we are going to retrieve all the sites in the region. We iterate over these sites to check whether the site is resolved or has a value other than zero. If it is, we then check the value associated with this site to see if it is different from our integer and continue analysing the region. If it is the same value, we return False.

4.2.7 IsCONNEX

IsConnex will compute the number of components in a graph. This rule was inspired and adapted by the algorithm "Depth-first search to find connected components in a graph" in the book "Algorithms Fourth Edition"[27].

For this rule, we will have a BitSet⁶ the size of the number of sites to be analysed and a linked list of integers. We will iterate over each cell. If the cell is not resolved and is not marked in the BitSet, or if the cell has a value other than the imposed value and is not marked, then we can consider it as marked and add this cell to our linked list. As long as this linked list is not empty, we will retrieve this node from the list and we will retrieve all its neighbours. If this neighbour meets the same conditions as above, then we consider it to be marked and add it to our linked list.

This gives us the number of components. If the number exceeds the authorised number, we return the boolean False.

⁵<https://www.nikoli.co.jp/en/puzzles/Hashiwokakero/>

⁶<https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/java/util/BitSet.html>

4.2.8 SUPER LUDEME AT

The super ludeme At has the same function as Is. This super ludeme refers to other ludemes that will be described here. As with Is, we have created types to structure the ludeme. We have only created the regionResult type, which checks the number of elements in a region (fig: 4.2).

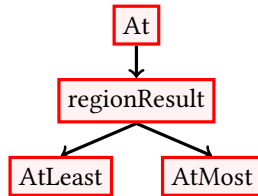


Figure 4.2: The super ludeme At

4.2.9 AtMost & AtLeast

AtMost is the first ludeme created with the At super ludeme. It was created in the same way as the IsSum ludeme. We will count the elements and add them together. The difference lies in the final condition. We no longer return False if we don't reach the desired value, but only if we exceed the desired value.

This rule has an individual attribute in its arguments. When we have this activated, we ensure that the values proposed to complete a region do not exceed the integer given as an argument.

AtLeast will work in exactly the same way, but in reverse. We want a minimum number of elements rather than a maximum. AtLeast does not have the individual attribute.

4.2.10 AllHintDifferent

AllHintDifferent is similar to AllDifferent. Like AllDifferent, the aim is to have different hints within a region. We will retrieve the hints from the region and iterate from site to site. If the site is unresolved, we will retrieve the hint present on it and store it in a list if the hint is not already there. If not, this would mean that we have the same hint twice, so we return the boolean False. If we only have different hints at the end of the iteration, then the rule has been respected and we will return True. This rule will be executed as an end rule with IsSolved.

4.3 THE NEW PUZZLES

These new puzzle include many variants of existing games, as well as new games that have required the addition of the conditions mentioned above, as well as graphic changes. These will be discussed in the next section of this chapter. Each puzzle has been created using the same ludemic representation as the previous games. Different challenges have also been added for each game. So there are different instances of the game with, for

example, different board sizes, different hints, etc. All the puzzles added have been found in the register of games created by Nikoli⁷, or in the Cross+A⁸ software catalogue. These new puzzles have been developed with two aims in mind: to complete the Ludii system's catalogue of deduction puzzles, while covering as many puzzle types as possible according to the taxonomy[15].

4.3.1 THE SUDOKU VARIANTS

This sub-section will discuss of the ten new Sudoku variants available. So many variants have been made possible thanks to the ludemic approach. The general aspect of these rules means that many puzzles can be created without having to add new elements. Take Sudoku, for example. We have created almost a dozen variants of this puzzle based on its basic model. We have been able to reuse the same graphic style for a "classic" board, but also the same rules for all these variants. Of course, some modifications are necessary to make these variants unique and lie in the graphic representation of the puzzle itself and not in the rules. Another approach would have made development less practical, as we would probably have had to start from scratch, as quoted in [24]. It also gave us a better insight into the Ludii system. All these puzzles were found in the Cross+A⁹ catalogue.

CENTER DOT SUDOKU

The Center Dot Sudoku¹⁰ (fig: 4.3) is the first Sudoku variant created in this project. We were able to reuse the ludemic representation of classic Sudoku. The rules remain the same, except that this Sudoku have a new region which is the center of each sub-grid. You have to fill in the grid with numbers so that all the regions (columns, rows, sub-grids and the center of each sub-grid) are filled in. To make the game more visual, we have added a colour to the sites in the new region. As with Sudoku, some of the numbers are already positioned.

To represent the puzzle, the classic nine-by-nine board is used. The user can place numbers from one to nine. The rows, columns and sub-grids regions were generated automatically using the keywords "Columns Rows SubGrids". These keywords will generate regions. Rows will generate one region per row, Columns will do the same per column. SubGrids will generate one region per sub-grid of size nine by nine, as in a classic Sudoku. The region with the centers of the sub-grids was added manually using the cell indexes. The sub-grids were designed in the same way as for classic Sudoku, using the Sudoku style that had already been created.

For the condition, only the AllDifferent ludeme is applied. As mentioned above, it is applied to each region of the game to prevent players from placing the same number twice.

⁷<https://www.nikoli.co.jp/en/puzzles/>

⁸<https://www.cross-plus-a.com/fr/index.htm>

⁹<https://www.cross-plus-a.com/fr/index.htm>

¹⁰<https://www.cross-plus-a.com/fr/sudoku.htm#CenterDot>

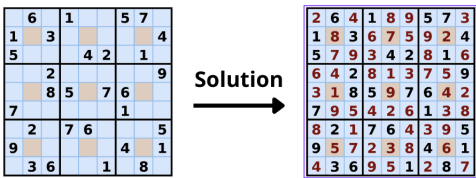


Figure 4.3: An Empty Center Dot Sudoku and its solution

WINDOKU

Windoku¹¹ (fig: 4.4) is also a variant of Sudoku. Windoku still have the same board structure, but this time the puzzle have not one but four new regions. These are four sub-grids separated by a row or column to represent a window. As with center dot Sudoku, the new regions have been coloured to make them clearer.

As with classic Sudoku, AllDifferent is the rule applied to this game.

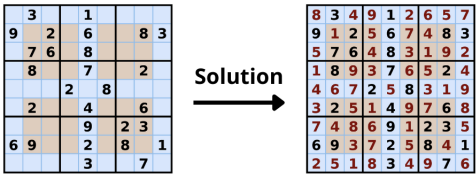


Figure 4.4: An Empty Windoku and its solution

JIGSAW

Jigsaw¹² (fig: 4.5) follows the same storyline as the previous games. The aim is to complete regions with numbers from one to nine. This puzzle also have the concept of rows and columns, which is defined for the regions, but we no longer have the sub-grids. The sub-grids are replaced by specially shaped regions. Numbers are also already placed to guide the player towards the solution.

From a gameplay point of view, the rows and columns are defined using the logical words Rows and Columns. For the nine other regions that replace the sub-grids, we have to define them by hand within the game descriptions.

For the ludemes used, we only have the AllDifferent called. From a graphical point of view, the regions are defined as HintRegions because they are regions that are drawn for the user. Before, only the contours of hint regions were drawn (if we called them).

¹¹<https://www.cross-plus-a.com/fr/sudoku.htm#Windoku>

¹²<https://www.cross-plus-a.com/fr/sudoku.htm#JigsawSudoku>

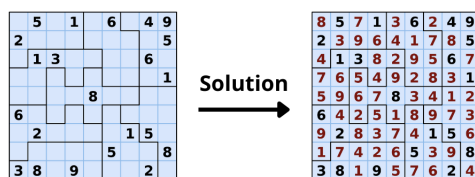


Figure 4.5: An Empty Jigsaw and its solution

ASTERISK SUDOKU

As the name suggests, this is also a variant of Sudoku. Asterisk Sudoku¹³ (fig: 4.6) have a classic board with regions that are the classic rows, columns and sub-grids. In addition to these regions, the puzzle have a new one. This corresponds to the center of the board and eight other cells surrounding it.

For the representation of this game, this Sudoku have a classic representation as before with the keywords Rows, Columns and SubGrid. The new region had to be defined by hand using the indexes of these new cells. These cells have been coloured to make the game clearer for the user.

As for the rules, there is not much difference from the others, as we are still using the same rule just different regions.

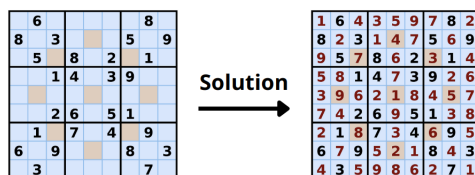


Figure 4.6: An Empty Asterisk Sudoku and its solution

GIRANDOLA

Although it doesn't have Sudoku in its name, Girandola¹⁴ (fig: 4.7) is a variant of Sudoku too. The board is still a nine by nine square with rows, columns and sub-grids as regions. The challenge lies in the new region that has been created. These are located in the four corners of the board and in the center of the other sub-grids.

The representation remains the same as the other Sudoku puzzles, but the new region has been defined manually. It has been coloured so that the player can see the new region.

As before, only one rule has been used: the AllDifferent.

¹³<https://www.cross-plus-a.com/fr/sudoku.htm#Asterisk>

¹⁴<https://www.cross-plus-a.com/fr/sudoku.htm#Girandola>

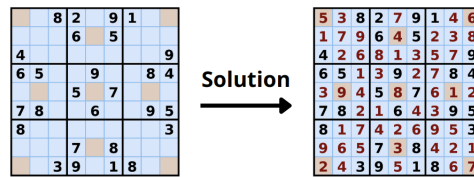


Figure 4.7: An Empty Girandola and its solution

SUDOKU DG

Sudoku DG¹⁵ (fig: 4.8) is the colourful version of Sudoku. In this version, we have a nine by nine square board made up of classic sub-grids. On top of that, we have nine new regions. In each sub-grid we will take the first cell to make a first region, then the second cell of each sub-grid to make the second, and so on. Each region will have a different colour to visualise each new region.

When it comes to representing the game, everything remains as classic as in previous puzzles. We need to define the nine regions by typing in the cell indexes to create them. As mentioned, each region will have its own colour to differentiate them.

For the rules, only the AllDifferent is used, so that the same number cannot be used in the same region.

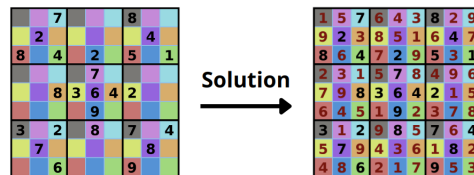


Figure 4.8: An Empty Sudoku DG and its solution

FLOWER SUDOKU

This and the following variants have specially shaped board. In this first variant¹⁶ (fig: 4.9), the game board consists of a central Sudoku with a new row of three sub-grids at each cardinal point. The aim is to make the board look like a flower. In this Sudoku, there is not one but five Sudoku to solve. There is one in the east, one in the west, one in the north, one in the south and one in the center. The difficulty lies in the fact that some Sudoku have sub-grids in common.

To represent this game, we have merged the boards. The first board is the central board, measuring nine by nine. We then generated four other boards that we positioned at specific coordinates to represent our flower. Compared with the previous Sudoku game, we can no longer use keywords because we have an unusual board. We had to define each region manually. Each row of nine elements, each column of nine elements and each sub-grid was

¹⁵<https://www.cross-plus-a.com/fr/sudoku.htm#SudokuDG>

¹⁶<https://www.cross-plus-a.com/fr/sudoku.htm#FlowerSudoku>

manually encoded to make the puzzle work. As the board is special, each edge to represent the Sudoku was also added manually using graphical metadata.

The rules remain classic. Despite the special shape of the board, the only rule used is AllDifferent.

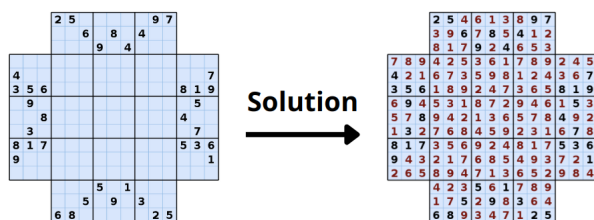


Figure 4.9: An Empty Flower Sudoku and its solution

KAZAGURUMA

Kazaguruma¹⁷ (fig: 4.10) is a Sudoku variant that also has its own special board. In this puzzle, the board is made up of five combined Sudoku to present a propeller of mill. The aim of the puzzle is to solve these five Sudoku simultaneously, i.e. taking account of each other as some of the sub-grids are shared.

The puzzle has been created in the same way as Flower Sudoku, i.e. with a central board and then four other boards arranged so that our shape appears. The central board has been coloured to make it stand out. Like our previous variant, we had to encode all the regions manually to make the puzzle work. The edges also had to be manually added.

For the rules, only the AllDifferent is called.

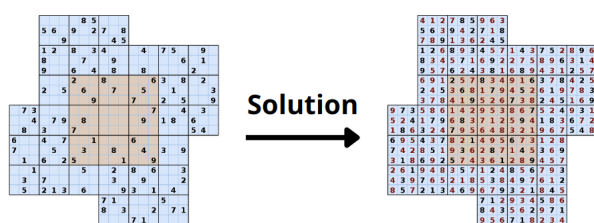


Figure 4.10: An Empty Kazaguruma and its solution

SOHEI SUDOKU

Sohei Sudoku¹⁸ (fig: 4.11) may seem simpler than its predecessors, as there are only four Sudoku to solve. For this Sudoku, only four sub-grids are shared, which limits the possibilities and makes it easier to make mistakes.

¹⁷<https://www.cross-plus-a.com/fr/sudoku.htm#Kazaguruma>

¹⁸<https://www.cross-plus-a.com/fr/sudoku.htm#SoheiSudoku>

To represent this puzzle, four boards have been created and merged to create the board. The large central cell was removed by creating a polygon using the indexes of this cell. As for the regions, they had to be added by hand so that each region corresponded correctly.

For the rules, as with the other Sudoku so far, only AllDifferent was used.

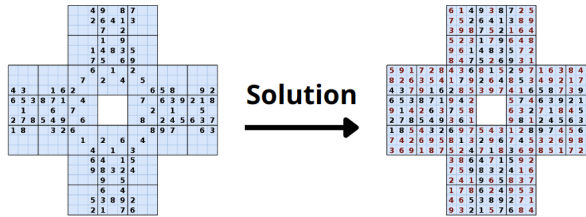


Figure 4.11: An Empty Sohei Sudoku and its solution

BUTTERFLY SUDOKU

Butterfly Sudoku¹⁹ (fig: 4.12) is not a nine by nine Sudoku, but a twelve by twelve Sudoku. There are not one but four Sudoku to solve on this board. The first is in the North-West corner, the second in the North-East corner, the third in the South-East corner and the last in the South-West corner.

Representing this puzzle was simpler than the previous ones, as we were able to create a twelve by twelve square and automatically draw the sub-grids in Sudoku style. Only the regions had to be encoded manually. We can't use the classic keywords because that would create regions for each row and each column of twelve elements rather than nine. Some of the sub-grids have been coloured to make it easier to solve the puzzle. The green sub-grids are common to all four Sudoku and the orange sub-grids are common to two Sudoku.

AllDifferent is the only rule needed to solve this puzzle.

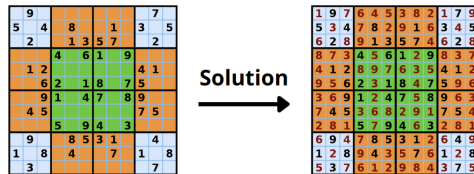


Figure 4.12: An Empty Butterfly Sudoku and its solution

4.3.2 OTHER PUZZLES

From this sub-section onwards, we will be discussing the new puzzles that have been added. These have been selected from the catalogues with a view to modelling them in the Ludii system. These puzzles have been chosen to offer new mechanics but also to try and cover as much of the taxonomy[15] as possible.

¹⁹<https://www.cross-plus-a.com/fr/sudoku.htm#ButterflySudoku>

4.3.3 AKARI

Akari²⁰ (fig: 4.13) is a puzzle created by the Japanese firm Nikoli in 2001. In this game, we have a square board of a certain size. On this board, there are some blackened squares, some with numbers and some without. The aim is to light up all the squares that are not black. To do this, we are going to place light bulbs according to certain rules. Around the blackened squares with a number on them, we must have the exact number of bulbs. Only one light bulb can be placed in a row or column. Finally, the blackened squares (with or without a hint) are like walls and therefore divide the rows. Logically, no bulbs can be placed on these squares.

4

To represent the puzzle, the board is a square of predefined size. On this puzzle there are hints which are designated using the Hints objects. Three different regions have been created: the "lines", which represent each line where the player can play, the "walls", which represent all the blackened squares without numbers on them, and the "hints", which are the blackened squares with a number written on them. From a logical point of view, the player will place one's. For the graphic and playful side of the puzzle, the one's are replaced by a light bulb for the user. The blackened squares have been greyed out to make them stand out. All this has been achieved using graphical metadata.

The puzzle conditions are made up of a set of ludemes. First of all, IsSum is applied on the Walls region so that no light bulbs can be placed. The sum must therefore be zero. Then the AtMost ludeme is called on each "line" region. So we have a maximum of one light bulb per region. The condition will then iterate over each hint using a ForAll Hint. This notation will allow us to move from hint to hint and apply a ludeme to each of them. On each one will be applied an IsCount to have the right number of light bulbs around these sites. The regions given as arguments to IsCount are generated using the Sites Around ludeme. This ludeme creates a region by taking the adjacent squares orthogonally to the square we are on. Finally, it is mandatory that there is at least one light bulb per region. To do this, each cell will be checked with a ForAll cell to which AtLeast will be applied with the number one to ensure that at least one bulb is in the region. The region given as an argument is created using the Sites Direction ludeme. It will generate a region starting from the cell we are in, starting orthogonally. The region stops if we encounter a blackened cell.

What's new about this puzzle is that we will need a minimum or maximum number of pieces in a region, rather than a precise number. This puzzle is part of the symbol-type puzzles described in the taxonomy[15].

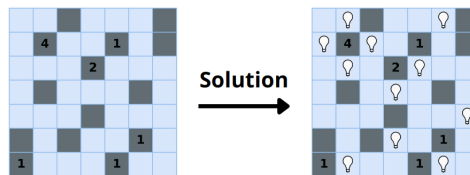


Figure 4.13: An Empty Akari and its solution

²⁰<https://www.nikoli.co.jp/en/puzzles/akari/>

4.3.4 HITORI

Hitori²¹ (fig: 4.14) is a Japanese puzzle diffused by Nikoli. In this puzzle, you need to block certain squares so that each values in each row and column are different. If a square is blocked, you can not block an adjacent one (orthogonally, diagonals are allowed). The remaining squares must form a single block. You can not divide them into two groups, for example.

To represent this puzzle, nothing could be simpler: the board is a square of a certain size with numbers written on it. To place these numbers, hints object are used so that other number can be place on them. If we set numbers in the starting rules, the square would be considered solved and therefore not modifiable in the eyes of the system.

For the puzzle conditions, the first is IsConnex. We have not applied any specific numbers to the rule, as we are putting ones on the plate, which is the base number in the rule. Two ForAll is used to over the cells. AtMost is applied to checks that two values are not blocked if these two values are arranged orthogonally. To generate this region, the Sites Around ludeme is used. Finally, in the end rules, the usual IsSolved rule is applied with the ludeme AllHintDifferent to check that for each row and each column the value are different. This rule must be executed at the end of the game, because if we execute it during the game, the system will consider each move to be false. In effect, the solution would be under construction.

This new puzzle has been added because it is one of the shading puzzles mentioned in the taxonomy[15]. These puzzles were not yet present in the Ludii system.

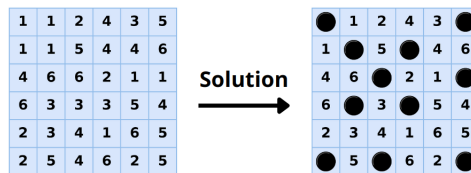


Figure 4.14: An Empty Hitori and its solution

4.3.5 RIPPLE EFFECT

Ripple Effect²² (fig: 4.15) is a number puzzle diffused by Nikoli. It is a square board where you have to complete the regions. These regions range in size from a minimum of one up to a certain size. In the regions, you have to fill in the squares according to the size of the region. If the region has a size of one, we need to fill in the number one. If the region has a size of three, yo need to fill in the numbers one, two and three and so on. You need to make sure that the same numbers are separated by a distance equivalent to that number, which makes the puzzle more complex.

This puzzle is represented using a square board of a certain size. The regions are defined using hints. The associated hints are the size of the region. This allows us to define the

²¹<https://www.nikoli.co.jp/en/puzzles/hitori/>

²²https://www.nikoli.co.jp/en/puzzles/ripple_Effect/

regions in addition to using the size of the region in our rules. The Ripple Effect has its own particular graphic style.

For the conditions, the first condition is AllDifferent to ensure that the numbers in a region are all different. Then the AtMost ludeme is called with the individual attribute. This will ensure that the number proposed does not exceed the size of the region. Finally, each cell will be visited in order to apply the IsDistinct ludeme with a certain region and an integer equivalent to a value associated with the cell. The region is created using the Sites Direction ludeme to retrieve all the cells orthogonally within a radius the size of the value associated with the cell. This prevents the player from setting unauthorised numbers based on his neighbours.

This puzzle has been modelled to develop a new mechanic: blocking numbers in a region. This puzzle is part of the puzzle symbol category of the taxonomy[15].

4

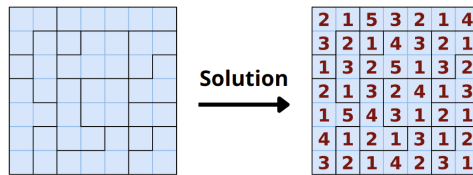


Figure 4.15: An Empty Ripple Effect and its solution

4.3.6 HASHIWOKAKERO

Hashiwokakero²³ (fig: 4.16) is a Nikoli game. In this puzzle, you try to connect numbers with the exact number of links needed to match a hint. The links are either one or two. You can think of them as numbers on a blank page and we want to connect them all vertically or horizontally to form one group.

To represent this new puzzle, a graph is developed. To do this, each vertex must be defined using its coordinates. The starting point of the graph is at the bottom left. The first vertex of the graph will have index zero, the second index one and so on. Then, each possible link must also be defined using the index of the vertices it connects. Once the graph has been created, the hints will be added to each vertex. A specific graphic style inspired by the PenAndPaper style has been created for this puzzle. This style removes the cells from the display, leaving only the vertices and edges.

Two rules will apply to ensure that the game runs smoothly. Firstly, the links must not cross. To do this, the IsCrossed ludeme with the keyword not is used. This will ensure that the links do not cross. Then, all the vertices will be visited to check that the right number of links are associated with them. This is done using the IsSum ludeme. The link region is retrieved using the Sites Incident ludeme, which will retrieve all the links around a vertex. Finally, at the end of the game, in the end rules, we check that we only have one group. This rule will be executed as an end rule to allow the player to play in various places to determine pieces of solutions and then assemble them.

²³<https://www.nikoli.co.jp/en/puzzles/hashiwokakero/>

Hashiwokakero was an interesting puzzle to model because, as well as adding to the catalogue of puzzles modelled in Ludii, it offered a new mechanic with edges that didn't involve building a path. This puzzle is in the Path Connection puzzle category in the taxonomy[15].

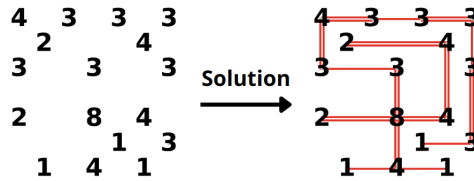


Figure 4.16: An Empty Hashiwokakero and its solution

4.3.7 NONOGRAM

Nonogram²⁴ (fig: 4.17,) also known as Picross, is a game where you have to colour in squares to make a picture appear. The squares to be coloured are indicated on the left-hand side of the puzzle for the rows and on the top of the board for the columns. The board to be completed is either a rectangle or a square of a defined size.

To represent this puzzle, the first step was to modify all the logic associated with hints in the system. Previously it was only possible to have one hint per region. Now it is possible to have several per region. To do this, we had to create a new method and a new constructor in the Hint class. It is now possible to store several hints in an integer array. This also required a number of changes to the equipment, context, etc. in order to evolve the data structures so that they could correspond to having several hints. The ludemes using hints directly also had to be updated as a result of these new changes. The method for representing hints has also had to be updated, but we will discuss this again in the graphics section.

The board is shaped like a rectangle of predefined size. The hints are also defined for each row and column. As in previous games, the player will have either zeros to avoid selecting the square or ones to select it from a logical point of view. In the eyes of the user, it is not zeros that will appear but white crosses and it is not ones either but black squares.

For the conditions, only IsMatch is called. This ludeme will generate all the possibilities for each region in a combinatorial way.

The nonogram is a real challenge to model. This puzzle offers a new vision of hints while working regionally using the concept of regular expression. It is one of the shading category puzzles according to the taxonomy[15].

²⁴<https://fr.wikipedia.org/wiki/Picross>

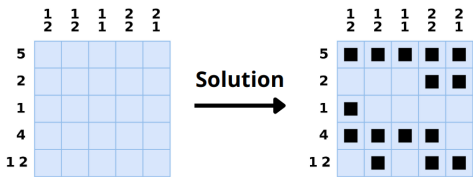


Figure 4.17: An Empty Nonogram and its solution

4.3.8 COLOR NONOGRAM

Colour Nonogram (fig: 4.18) is, as the name suggests, a variant of nonogram. The aim of the game and the rules remain the same, except that we no longer complete the board using black squares but squares of different colours.

The puzzle is represented using the same structure. In other words, the board is a rectangle with a defined size. The hints are also represented in almost the same way. They now have a new attribute, which is colour. We have determined a code for representing colours: an integer corresponds to a colour. Here is the list:

- 0 is a cross
- 1 is Black
- 2 is White
- 3 is Blue
- 4 is Red
- 5 is Yellow
- 6 is Green
- 7 is Orange
- 8 is Purple

This has necessitated new changes to the context and equipment, but these are minimal compared with the previous changes. The only changes made are to recover the integers and therefore the colours associated with the hint. From a graphical point of view, the representation of hints has been modified to include new colours. The player will position different integers to complete the puzzle. The zeros remain as crosses in the player’s eyes, while the other integers are squares of different colours.

For the conditions, the IsMatch is also called, but with its Color attribute. This will change the logic of the code, because it is no longer just zeros and ones that will be laid down, but different integers representing colours. This makes it possible to generate specific cases to check whether the solution is still achievable or not.

This puzzle is part of the shading puzzle category in the taxonomy[15]. This puzzle has been developed as a variant of the nonogram, which has been implemented earlier.

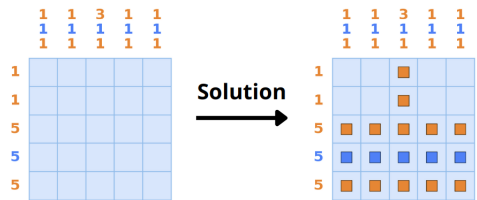


Figure 4.18: An Empty Color Nonogram and its solution

4.3.9 HEXAGONAL NONOGRAM

The Hexagonal Nonogram (fig: 4.19) is the second variant of the Nonogram. This puzzle is the same as the basic puzzle, except that the puzzle is no longer square but hexagonal.

For the representation, rectangles are no longer defined, but a hexagon of size X by X. The hints are represented in the same way as in the classic game, i.e. one or more hints for the same region. The hints no longer point North and West, but now indicate three directions: West, North-East and South-East. From a graphical point of view, the player still places zeros or ones for logic. Graphically for the player, the zeros are still crosses and the ones are hexagonal pieces rather than squares.

The IsMatch condition remains functional despite the change in shape, and we don't call up the colour attribute.

Like Nonogram, this puzzle is in the shading puzzle category. It uses the same mechanics as the Nonogram.

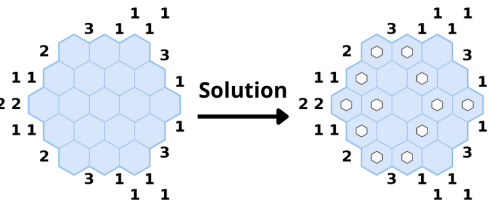


Figure 4.19: An Empty Hexagonal Nonogram and its solution

4.3.10 MASYU

Masyu²⁵ (fig: 4.20) is a puzzle where you have to complete a path. The aim of this path is to pass through all the integers on the board. You don't necessarily have to pass through all the vertices. Nikoli has added a difficulty to the puzzle by forcing players to make a certain move depending on the hint they encounter. If it is a zero, we have to make a straight line, and if it is a one, we have to make sure that the path is a bend.

This puzzle is represented by a rectangle on which we fill in the edges. This will act as the path. The hints are used to note the indices on the vertices. From a graphical point of view,

²⁵<https://www.nikoli.co.jp/en/puzzles/masyu/>

the PenAndPaper style has been used, which was useful for removing the background of the board and bringing out the edges and vertices.

For the conditions, the first is the `IsValidDirection` which has been created for this puzzle. Depending on the hint encountered, this will force the path created by the player to take a certain direction (straight ahead if we have a zero, a turn if we have a one). Each vertex will then be visited to count the number of edges linked to that vertex. Each vertex can be linked by either zero edges or two edges. We want to have either two edges in order to make the path and prevent the path from crossing. The second is zero, which allows solutions to be found without passing through all the vertices. Having two edges per vertex whenever we want ensures that we have a path and not a group of edges that will never meet. An iteration will also be carried out on the hints to check that the path passes through all the hints.

4

Like the Slitherlink, the Masyu is a path loop puzzle in the puzzle taxonomy[15]. The new mechanics associated with this puzzle are that the hints are found on vertices rather than in cells. These hints dictate the direction of the path.

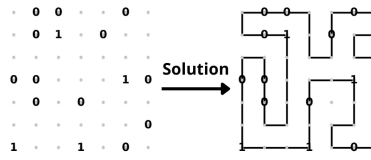


Figure 4.20: An Empty Masyu and its solution

4.3.11 KURODOKO

Kurodoko²⁶ (fig: 4.21) is a puzzle game from Nikoli. In this puzzle, you must place black blocks in order to block squares. The aim is to rely on the hints present on the board and create orthogonal regions that have a distance equal to the hint. Note that the square with the hint is taken into account when calculating the size of the region. Two black blocks cannot be laid adjacent to each other in an orthogonal direction (diagonals are allowed).

This puzzle is represented using a square of predefined size. From a logical point of view, these are some ones that will be installed. From a graphic point of view, these ones are transformed by black blocks. Hints will be represented by object hints. A "Walls" region will be created, containing all the cells containing a hint. This will make it possible, with the conditions, not to place a block on these cells.

For the conditions, each cell will be visited to check that two blocks are not blocked adjacent to each other. `IsSum` will be applied to the "Walls" region so that no blocks are placed on these cells. The number given to `IsSum` will therefore be zero. Each cell with a hint will be visited with the `IsCountEmpty` ludeme. This will check that in an orthogonal region, starting from the cell with the hint, the number of empty cells. This region will be generated using the `Sites Direction` ludeme.

²⁶<https://www.nikoli.co.jp/en/puzzles/kurodoko/>

This puzzle falls into the "position symbol" category, according to the taxonomy[15]. This puzzle uses the new mechanism of counting the number of empty cells.

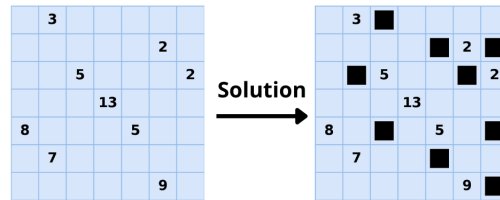


Figure 4.21: An Empty Kurodoko and its solution

4.3.12 USOWAN

4

Usowan²⁷ (fig: 4.22) is a game created by Nikoli. In this puzzle, black blocks must be placed to satisfy the hints. In some areas, there are real and fake hints. These blocks must be placed around the real hint. Among the rules: if there is a false hint, there can only be one in a region and it will be along an edge. All uncovered cells must form a single block (a single component).

To represent this puzzle, a square board of predefined size will be used. Hints will be represented by Hints objects and regions will also be represented. These regions are not defined with our Hints objects. The regions include several indices and are therefore defined manually. From a logical point of view, the player poses the number five. The number five has been chosen to differentiate it from the false hints, which will have numbers ranging from one to four. From a graphical point of view, these fives are replaced by black blocks.

At the start of the puzzle, numbers will already be placed on the board to simulate false hints. Among the condition during the game, IsSum will be applied to the region containing the real hints so that it cannot be manipulated. As the false indices are set, it is already impossible to play on these cells. Each hint will be visited to check the number of blocks placed orthogonally and adjacently around this hint. These blocks can be counted using the IsCount ludeme. Finally, uncovered cells are checked if they form a single component using the IsConnex ludeme.

Usowan is a shading-type puzzle in the puzzle taxonomy[15] and offers a new mechanic based on the fact that there are true and false hints.

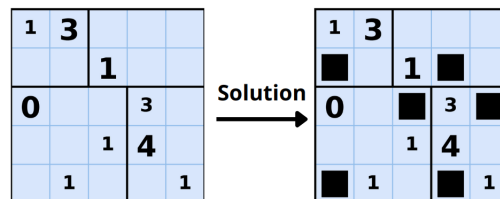


Figure 4.22: An Empty Usowan and its solution

²⁷<https://www.nikoli.co.jp/en/puzzles/usowan/>

4.3.13 BIG TOUR

The Big Tour²⁸ (fig: 4.23) is a puzzle found in the Cross+A²⁹ catalogue. The aim of this puzzle is to complete a path through all the vertices. Some links are already present on the board to restrict our movements. The path must be a non-intersecting loop.

The puzzle is represented using a rectangle. The player places edges on the board to form the path. The PenAndPaper graphic style is used to render the edges and vertices.

At the start, certain edges will be put down to force the player to make certain moves and find a solution. During the game, each vertex will be visited to ensure that two edges are connected to the vertex. This will ensure that each vertex is used and that there are no loops in the solution. The path can only be made up of a single loop, which is what will be checked.

This puzzle falls into the path loop category of the puzzle taxonomy[15]. Compared with other puzzles in this category, the path must pass through all the vertices.

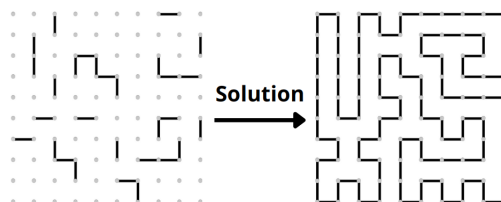


Figure 4.23: An Empty Big Tour and its solution

4.3.14 BURAITORAITO

In the Buraitoraito³⁰ (fig: 4.24) puzzle, we have to place stars. These stars are placed on squares not blackened with a number and must correspond to these hints. In the orthogonal region of these hints, you need to find the exact number of stars present on the hint.

To represent the puzzle, a square board is used. The player will have ones on the board, which will be represented by stars. Hints will be placed on the board with hints objects. A region containing all the hints will also be created so that the player cannot play on them.

For the puzzle conditions, each hint will be visited with the IsCount ludeme. This will count the number of stars in an orthogonal region starting from the hint cell. This region is obtained via the Sites Direction ludeme. It will stop when it encounters another hint or the edge of the board.

This puzzle bears a strong resemblance to Akari, as they are in the same category in the taxonomy[15]. The difference is that there is no longer a maximum of one symbol per region, but several defined by the hint.

²⁸<https://www.cross-plus-a.com/fr/puzzles.htm#GrandTour>

²⁹<https://www.cross-plus-a.com/fr/puzzles.htm>

³⁰<https://www.cross-plus-a.com/fr/puzzles.htm#Buraitoraito>

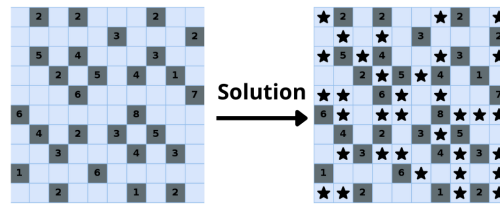


Figure 4.24: An Empty Buraitoraito and its solution

4.3.15 TILEPAINT

Tilepaint³¹ (fig: 4.25) is a Nikoli game. It can be seen as a cross between Nonogram and Kakuro. In effect, you have to colour in squares according to hints in order to make a drawing appear. In this puzzle, the hints are on the board itself rather than on the side. Tiles are present on the board. These tiles must be either completely colored or empty.

To represent this puzzle, a square board of predefined size is used. Hints are positioned using the cell indexes. Tiles are defined by regions, also using cell indexes. From a logical point of view, the player will place ones, but with the graphic metadata, he will place black blocks.

Among the condition, colored cells will be counted in each region with a hint. This is done using the IsSum ludeme. To validate the puzzle, the IsTilesComplete ludeme will be called in addition to the IsSolved ludeme. This will check that each tile is either empty or complete.

This shading-category puzzle is a combination of Kakuro and Nonogram mechanics. It adds tile mechanics.

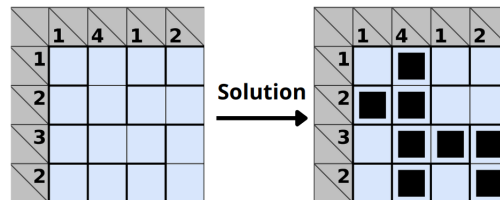


Figure 4.25: An Empty Tilepaint and its solution

4.4 GRAPHIC ADDITIONS

4.4.1 WHAT ALREADY EXISTS

Certain graphic styles are specific to certain puzzles. In general, a general graphic style is used, which can be modified with metadata. Metadata allow you to customize the graphic style of each puzzle. Below is the list of available metadata:

³¹<https://www.nikoli.co.jp/en/puzzles/tilepaint/>

BOARD STYLE

This metadata will call up a particular style of board. If this metadata is not called up, we call up the basic puzzle design. It is in this design that we are going to designate the outer and inner borders of the tray, as well as the filling of the background and the contents of the cells. We will also detect the hints and draw the regions that these hints comprise. The hints are also located in the design.

PLAYER COLOUR

Player Colour is a graphical metadata ludeme that assigns a colour to the player. This is used to differentiate between players in multi-player games.

4

PIECE RENAME

Piece Rename will allow us to change the content that the player will play. In most cases, the player will play numbers on the board, but we want them to have a particular shape depending on the puzzle. This metadata will allow us to change these pieces to the shape we want. (fig: 4.26 & 4.27)

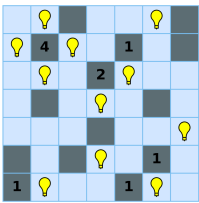


Figure 4.26: Akari with piece rename metadata

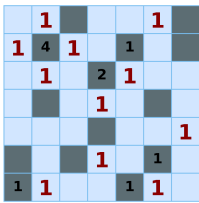


Figure 4.27: Akari without piece rename metadata

PIECE COLOUR

As with player colour, Piece Colour allows you to change the colour of the pieces you play. This metadata is preferable if you want to change the colour of several pieces. This metadata will allow us to change the colour of one piece, whereas the player colour will change the colour of all the pieces placed by the player. (fig: 4.28 & 4.29)

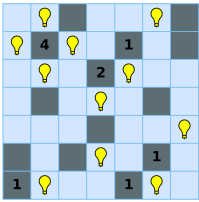


Figure 4.28: Akari with piece colour metadata

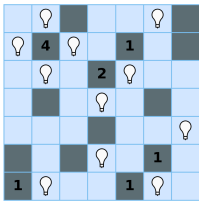


Figure 4.29: Akari without piece colour metadata

PIECE SCALE

It is sometimes necessary to change the size of the pieces to get a better rendering. To do this, we call the piece scale with a float. The size of the piece will be multiplied by the float given as an argument.

BOARD PLACEMENT

As with the pieces, it is sometimes necessary to modify the size of the board, particularly when we want to place information next to the board. To do this, we call the metadata Board Placement with a float to reduce or increase the size of the board.

DRAWHINT DIRECTION

The last metadata used is DrawHint, which takes a keyword indicating a direction as a parameter. This allows us to change the positions of the indices on the board. The 4 possible directions are Default (which is in the center of the square), TopLeft (which is the top left corner of the square), NextTo (which is on the side of the board) or None (which makes the clue invisible).

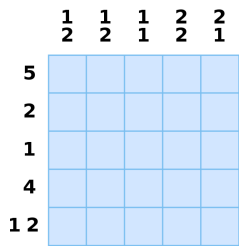


Figure 4.30: Nonogram with DrawHint metadata

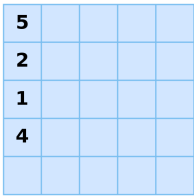


Figure 4.31: Nonogram without DrawHint metadata

4.4.2 CHANGES TO THE SYSTEM

To match the graphic styles of our puzzles as closely as possible, modifications had to be made to the existing styles.

DRAWHINT DIRECTION

Sometimes it is necessary to hide hints to make it easier to represent and model the puzzle. But these hints don't actually exist in the eyes of the player. To do this, we can use the DrawHint metadata with the None keyword. As this has not yet been implemented, it was done during this project. To do this, we simply skip the step of adding the hints to the board if the None keyword has been chosen.

THE HINTS

The hints have had to be modified to offer the opportunity to put more than one hint on the same region. As the structure had changed, we had to modify the retrieval of hints

for this to work. In addition, we had to add information to the "NextTo" keyword in the DrawHint graphical metadata to display several hints. To do this, we retrieved the same part of the code as for displaying a single hint, but we iterated over the list of hints and applied a distance between each one.

4.4.3 NEW STYLES

In addition to the metadata, it is possible to create a specific style for a game if required. Sudoku, for example, already has a special style that draws out the sub-grids, or Futoshiki, which removes the background and creates boxes for hints.

HASHIWOKAKERO

4

For the Hashiwokakero, the use of the "PenAndPaper" style was initially planned. The special feature of this puzzle is that the puzzle using single or double edges. We have added detection of the number played by the user, so we have doubled the link if the number is two. But this has caused a problem in other games. The changes we have made are not supposed to change the graphic style of other games in Ludii. In two-player games using this style, the rule is that player one places ones and player two places twos. The problem was that following our modification, player two was automatically placing double links, which no longer made sense in terms of representing the puzzle. We therefore created a style for the Hashiwokakero which is a derivative of the PenAndPaper style and which makes the double link according to the number encountered. This meant that we didn't have to change the style of the other games within the framework, but it also gave us the best possible representation of the Hashiwokakero.

HEXAGONAL NONOGRAM

The hexagonal nonogram needed a particular graphic style. Initially, we updated the Hints detection for this puzzle. The hints are no longer north and west, but west, north-east and south-east. Hints will be detected and classified according to the direction in which they point. This will allow the hints to be positioned in a certain way to represent the puzzle. Once this detection has been changed, the hints are visible. But a change, like the hashiwokakero, implies changes for the whole framework.

RIPPLE EFFECT

As far as the Ripple Effect is concerned, we have created a graphic style based on that of Sudoku. We have removed the drawing of the subgrids and, above all, we have changed the way the regions of our hints are drawn. In fact, in this puzzle, we have regions that must be visible to the user, and we have decided to implement them as hint regions to simplify our rules. So, in this unique style, we have modified the regions to have solid black lines rather than pink dotted lines.

TILEPAINT

In Tilepaint, as well as having the same look as Kakuro for our hints, we have tiles on the board. Since there are already hint regions, we need to add certain elements to detect our

non-hint regions. To do this, we have created a list of all these regions. We have created an auxiliary function that will keep all these regions. We will then draw them to represent these tiles. The style of this puzzle is therefore a derivate of Kakuro's style with all these new additions.

USOWAN

In this puzzle, we have regions that are not HintRegions. So we have created a graphic style that takes the detection of non-hint regions and their appearance on the board.

4.5 LUDEMEPLEX

Some puzzle descriptions use the same combination of ludemes. For example, Slitherlink and BigTour rules must take one and only one path without crossing each other. This rule is made possible by a combination of ludemes. As this combination is found in different representations, we decided to make it a ludemplex.

The definition given in Ludii's Game Logic Guide[21] is as follows: it is useful structures of component ludemes. These can be defined either:

- globally in their own .def file, in which case they become known defines, or
- locally within any .lud file in which they can be used.

Here are the 4 ludemplexes we have created:

1. OnlyOneWay: this ludemplex will run for path loop puzzles. It will check that the puzzle has been solved and that the path created is the only path (Masyu, Slitherlink, Big Tour).



Ludeme

```
1 (define "OnlyOneWay"
2   (and (is Solved) (= (count Groups Edge) 1))
3 )
```

2. EdgesByVertex: this combination of ludemes will ensure that on each vertex we have X edges to select. It is used for Big Tour puzzle.



Ludeme

```
1 (define "EdgeByVertex"
2   (forAll Vertex
3     (is Count Edge (sites Incident Edge of :Vertex
4       at:(from)) #1)
5   )
```

3. PieceNotAdjacent: this ludemplex will ensure that two pieces cannot be placed adjacent to each other. It is used for Hitori puzzle and Kurodoko Puzzle



Ludeme

```

1 (define "PieceNotAdjacent"
2   (forAll Cell (at Most (sites Around (from) N
3     includeSelf:True) 1 ))
4   (forAll Cell (at Most (sites Around (from) E
5     includeSelf:True) 1 ))
6 )

```

4. NumberOfEdgeByVertex: this last ludemplex will ensure that on each vertex we have either X(#1) edges to select or Y(#2). it is used for Masyu and Slitherlink puzzle



Ludeme

```

1 (define "NumberOfEdgeByVertex"
2   (forAll Vertex
3     (or
4       (is Count Edge (sites Incident Edge
5         of:Vertex at:(from)) #1)
6       (is Count Edge (sites Incident Edge
7         of:Vertex at:(from)) #2)
8     )
9 )

```

5

TESTING AND EXPERIMENTATION

In this chapter, we will discuss the various tests we have put in place to check that our work is working properly. We will then discuss the various experiments we have carried out on our puzzles. Every test and experiment was carried out on my own machine. This one is composed of an Intel Evo i5 processor, 16Gb RAM and a graphics card "Intel(R) Iris(R) Xe". The OS used is a Linux Ubuntu distribution.

5

5.1 THE TESTS

To complete this work, we had to use various tests. We used a test provided to us to check the integrity of the puzzles. We also created JUnitTests to check that the ludemes were working properly.

These different tests are essential to the project. Given that ludemes are objects that can be modified and improved, we need to be able to determine whether a ludeme has become false after modification. These different tests will therefore determine whether a puzzle is no longer solvable or whether a ludeme no longer responds to the different situations with which it is tested.

5.1.1 INTEGRITY TEST

This test will be run for each deduction puzzle we have in the system. To do this, each puzzle will be launched, and a trial will be run. These trials are a succession of movements that will be applied to the game to reach the final state. The trial will check that each movement applied is a legal movement. This is verified by all the rules in the puzzle. These tests are essential for the evolution of the games, as the test will show us which rule is blocking the puzzle. In this way, we can see whether a modification or development in our rules has compromised our puzzles. As well as being able to be run manually, this test will be run via GitHub Actions. Our project is stored on a public GitHub¹ and thanks to these actions, after each push, we will have this test running in addition to others. As well as seeing the integrity of the puzzles, this allowed us to see that we had not compromised the

¹https://github.com/Cailloux2123/Ludii_Memoire

rest of the system. In particular, when we created a puzzle, we were able to see that its name had been duplicated and had therefore rendered the system obsolete.

5.1.2 THE JUNITTESTS

JUnitTests were coded to check the accuracy of our ludemes according to certain scenarios. To carry out these tests, we used the TDD (Test Driven Development) method[4]. This method involves writing tests before coding our rules. We coded our scenarios to find out whether we wanted the ludemes to return True or False. Once the tests were implemented, we were able to work on the code of the ludemes so that the various tests corresponded to our ludemes.

In order to make the JUnitTests functional, we had to create a game description to test our game. This representation will be a five by five square board made up of a region. Trials are created to represent the different scenarios. To test the gameplay, we will launch the game with one of these trials (scenarios). We will apply each of the movements (even illegal ones) to the puzzle. We will create a ludeme object to check whether its method returns the expected boolean with the puzzle context. This context contains the puzzle after the various movements applied. We will compare this boolean with the boolean expected in the scenario to check whether the game system reacts as expected.

5

5.2 EXPERIMENTS

We run three different experiments. We are going to analyse the size of the challenges and the number of possible moves in a given time to see the system's limitations. We will also look at the number of tokens present in our game representation. A token is an element/word in the representation. Parentheses, brackets and other separators are not included in tokens.

The aim of these various experiments is to determine whether a ludemic representation can be considered effective. This means that the puzzle is displayed quickly and that the legal moves are quickly found. These different experiments will therefore highlight the puzzles that need to be optimised and highlight areas for improvement.

5.2.1 THE SIZE OF THE CHALLENGE

For these experiments, we have added challenges to some of our puzzles. Challenges are new instances of a puzzle. These instances do not change the rules or the way a puzzle works. They allow you to customise them by changing the numbers setup, the hints on the board or, in our case, the size of the challenge. With the help of these experiments, we want to see the limits of our puzzles in the system. To do this, we are going to add puzzles of different sizes to observe and understand the limits.

To check this, we are going to look at the NQueens puzzle, which is very simple in its representation, and the Nonogram, which has a complex ludeme: IsMatch.

Name And Size	Execution Time (s)
NQueens 5x5	0.29
NQueens 10x10	0.35
NQueens 15X15	0.99
NQueens 20x20	2.94
NQueens 25x25	6.6

For NQueens, we will see that sizes ranging from 5x5 to 15x15 are displayed very quickly and that the various movements are also displayed fairly quickly. From instance size 20x20, the display appears less quickly, as do the moves. From instance size 25x25 onwards, the graphics are really slow, and the system struggles to detect our movements. This is due to the radial pre-computation, which we will analyse later.

Name And Size	Execution Time (s)
Nonogram 5x5	0.21
Nonogram 10x10	0.68
Nonogram 15X15	9.86
Nonogram 20x20	X

For the Nonogram, we can see that the 5x5 instance is very simple and runs very quickly. We have the same observation for the 10x10 instance. The 15x15 instance, on the other hand, takes far too long and is therefore difficult to play. The 20x20 instance does not appear and causes system slowdowns. This is due to the IsMatch ludeme in the puzzle.

5.2.2 THE NUMBER OF MOVEMENTS IN 40 SECONDS

The aim of this experiment is to compare whether legal moves can be generated efficiently in the Ludii system. This will help determine whether a puzzle is not efficient enough. To do this, we will launch an instance of a puzzle (a puzzle challenge). For Nonogram, for example, it is a nonogram of different sizes with different hints. In the case of Sudoku, it is a grid with different starting numbers. For each instance of a puzzle, we will launch the game and make random moves for forty seconds. We will compare the number of moves made during the forty seconds according to the instance. It is important to note during the test that if, following random movements, we reach a final state of the puzzle (i.e. solved), we will restart the puzzle. We carried out the test taking into account:

- The execution time.
- The number of solutions found.
- The number of moves made during 40 seconds.
- The average number of moves per second during the execution time.

All results can be found in Appendix A (7.1). In this section, results relating to the NQueens puzzle and the Magic Square puzzle will be presented, as they present the same scenario as the other puzzles. The Nonogram puzzle will also be presented, as it presents a special feature.

Name And Size	Execution Time (s)	$N^{br}OfMovements$	$AverageN^{br}OfMovements$
MagicSquare 4x4	40	47,859	1167.61
MagicSquare 5x5	40	13,749	342.70
MagicSquare 10x10	40	186	5.07
MagicSquare 15x15	40	17	0.46

Name And Size	Execution Time (s)	$N^{br}OfMovements$	$AverageN^{br}OfMovements$
NQueens 4x4	40	263,037	6522.68
NQueens 5x5	40	94,595	2347.83
NQueens 10x10	40	2,716	64.14
NQueens 15x15	40	257	6.17

The following observations can be made: the larger the plateau, the fewer the number of movements performed. This observation is common to all puzzles. It can be explained by the Ludii system itself. When a puzzle is generated, radial precalculations are performed, as described in Chapter 2. The larger the board, the more calculations there are before the puzzle is ready. This justifies the smaller number of moves made on the larger puzzles.

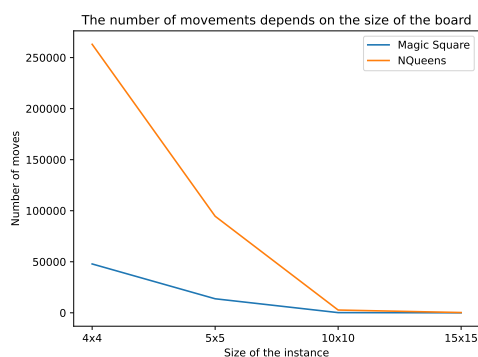


Figure 5.1: The number of movements depends on the size of the board for the Magic Square and the NQueens

A second observation is that for the same size, the number of movements performed is sometimes very different. This can be explained by the conditions given to the puzzle. The more calculations there are in the puzzle, the more moves are required to generate the various legal moves.

Name And Size	Execution Time (s)	$N^{br}OfMovements$	$AverageN^{br}OfMovements$
Nonogram 5x5	40	75,503	1903.003
Nonogram 10x10	40	573	14.83
Nonogram 15x15	46.22	7	0.159
Nonogram 20x20	1208.42	1	8.49E-4

We can see that the larger the Nonogram, the longer it takes to make a movement. When we have a Nonogram of size 5 by 5, the system can make 75,503 movements in forty seconds. When we have a Nonogram of size 10 by 10, we only have 573 movements made. That is 131 times less movement. We never see such a difference in movement in other puzzles. Nonograms of size 20 by 20 are a very different case. This one takes around 1200 seconds, i.e. twenty minutes, to complete a single movement. The problem comes from the gameplay in IsMatch. This ludeme will generate all the possibilities for a region in a combinatorial way. If at least one solution is found (i.e. matches the hints) then the puzzle will return True. We have tried to keep the main idea of this puzzle by generating cases and using regular expression to check our hints and cases. One optimisation we propose is to limit this search. To do this, we will check whether we have several hints on the same region. If this is the case, we will only generate cases where the start of our case corresponds to our first hint. This will allow us to limit the search by reducing the number of cases generated and explored. Here are the numbers after optimisation:

5

Name And Size	Execution Time (s)	$N^{br}OfMovements$	$AverageN^{br}OfMovements$
Nonogram 5x5	40	111,377	2784.425
Nonogram 10x10	40	787	19.675
Nonogram 15x15	42.498	9	0.225
Nonogram 20x20	948.91	1	0.025

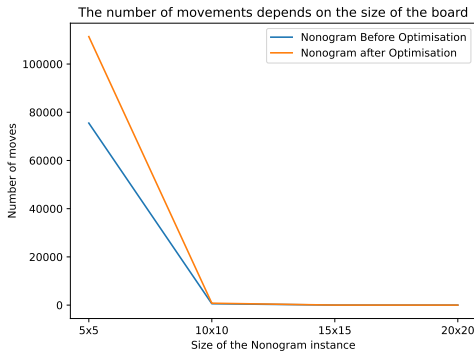


Figure 5.2: Number of move during 40 seconds by instance

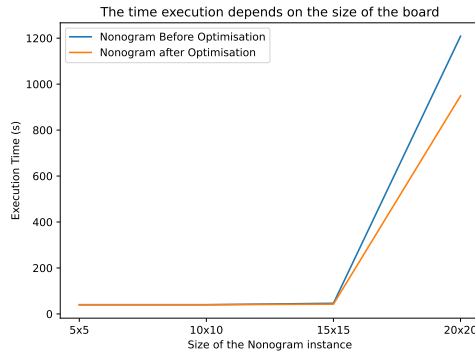


Figure 5.3: time execution by instance

5

We can see that for the 5 by 5 Nonogram, we have more or less 111,000 movements during the forty seconds. For the other instances, we still have a big drop in movement, but we are still seeing improvements. The most remarkable difference is for the 20 by 20 nonogram. It is still unusable at the moment because it still takes too long to calculate all the cases, but we have saved a total of 250 seconds on puzzle generation. To be sure, we would still have to optimise the puzzle by changing the logic of the puzzle and no longer generating the cases, but this would make the regular expressions unusable. As shown by [3], solving the Nonogram is very complex. A human solving the game will not take into account all the information on the board, whereas an algorithm will have to generate and combine everything. So when we have bigger puzzles, we need to do a lot more calculations. There is an additional difficulty. In the task of creating and modelling a puzzle, we don't want to solve the puzzle, we want to offer the players the moves they are allowed to make. We don't generate the moves that give a solution, but all the moves that don't go against the rules. The Ludii system will not try to find a solution by proposing a single path. It will explore all the moves in order to propose all the possible legal moves. This is much more complex than offering a single resolution path.

5.2.3 ANALYSIS AND SIMILARITY

The first two experiments (the size of the challenge and the number of movements in forty seconds) propose similarities in their results. The longer an instance takes to load, the fewer movements the test can perform. This can be explained by two factors that we have already mentioned:

- Radials: this pre-calculation, carried out when the puzzle is generated, calculates the position of the squares in the same regions as them to facilitate certain movements.
- The efficiency of a ludeme, which will take more or less time to execute and complete its task.

These two factors limit the display and use of certain puzzle sizes. That is why it is imperative that our games are as effective as possible.

This paper [5] proposes a different approach to solving the Nonogram. In the solver they have developed, they consider the Nonogram to be a single line. They add new information to a PriorityQueue (which represents this single line) in order to find new solutions. They then use probing algorithms to find the solutions to the puzzle in their search tree. Ludii could take inspiration from this solution by representing the Nonogram differently. Currently, the different rows and columns are compared to regular expressions in order to create the movements allowed in the puzzle and determine when a solution is found. Using another representation, it would be possible to determine the different constraints of the puzzle instance (based on its hints) and determine the legal moves. Ludii, unlike the solver developed in [5], will generate all the legal moves using the different constraints. It is the player who will use these moves to find a solution. Once a solver has been implemented, it will retrieve the various constraints (ludemes) in order to find a solution efficiently. But for this to be possible, we need to optimise the representation of these constraints, including those of the nonogram which, as the figures show, take too long to calculate and are therefore available to the player or a possible solver.

5.2.4 THE NUMBER OF TOKENS PER REPRESENTATION

The number of tokens shows whether or not a ludemic representation is clear. This test will count the number of tokens in a ludemic representation of a puzzle. The results of this experiment can be found in Appendix B (7.2). Here are some numbers:

Name	$N^{br}OfTokens$
Latin Square	30
Sudoku	74
Center Dot Sudoku	105
Flower Sudoku	1007
Butterfly Sudoku	1059
Kazaguruma	1602

The shorter the description, the more understandable it will be. That is why every description should be as short as possible. Looking at the figures (fig: 5.4), puzzles like Latin Square are really simple to understand because few tokens are used. But these numbers can be quite high, as in the Kazaguruma puzzle. This is justified by the number of elements to be defined for a puzzle. Kazaguruma uses a very large board in which you have to define a lot of regions manually. This justifies its large number of tokens.

A second observation confirms the first. In Sudoku, seventy-four tokens are used. For these variants, new elements need to be implemented. Center Dot Sudoku, for example, has a new region compared with classic Sudoku. This is why the puzzle has one hundred and five tokens. In the bigger variants with larger boards and therefore more regions, such as Flower Sudoku or Butterfly Sudoku, there are many more elements to define. This explains why these two puzzles use more than a thousand tokens.

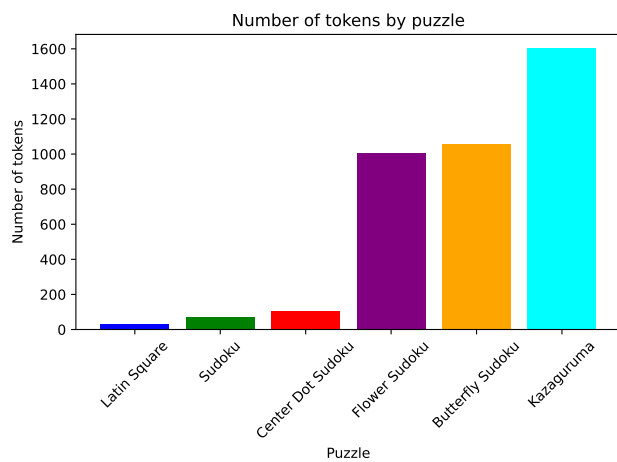


Figure 5.4: Number of tokens by puzzle

6

FUTURE WORK IN CONNECTION WITH THIS THESIS

6

6.1 GENERAL GAME PLAYING AND CONSTRAINT PROGRAMMING TO SOLVE ANY LOGIC PUZZLES: TOM DOUMONT'S THESIS

Our part with the deduction puzzles was to create and model them. So we took care of the logic with the game equipment and the rules to make it work. Another interesting point should be the solving of these puzzles. This would be done using a solver and XCSP constraints[20]. This research work is being carried out by another student, Tom Doumont. This work will retrieve our new ludemes and convert them into XCSP constraints. This will enable the puzzles in Ludii to be solved in different solvers like Choco¹ or Ace². Oscar³ and minicp⁴ should also be used. The answers provided by the solver can be reused in Ludii to create AIs to solve puzzles.

6.2 POSSIBLE FUTURE WORK

6.2.1 ADDITION OF MANY OTHER GAMES AND NEW RULES

Looking at the taxonomy of the puzzles[15], we can see that we worked with different categories. Deduction puzzles fall into the abstract category of this taxonomy. Before our additions, we only had puzzles in the following categories:

- Position with Symbols like Sudoku (fig: 3.1), Kakuro (fig: 3.3) or NQueens (fig: 3.6)
- Path with Loop for Slitherlink(fig: 3.9). It's the only one of its kind.

¹<https://choco-solver.org/>

²<https://www.cril.univ-artois.fr/en/software/ace/>

³<https://webperso.info.ucl.ac.be/~pschaus/oscar.html>

⁴<http://www.minicp.org/>

In the following figures (fig: 6.1 & 6.2 & 6.3 & 6.4 & 6.5), the blue nodes represent the puzzles and categories already in the Ludii system. In red are all our contributions.

After the additions, the different puzzles cover five categories. We have completed the two existing categories with:

- Position with Symbols with, for example Centre Dot Sudoku, Akari and Buraitoraito (fig: 6.1).

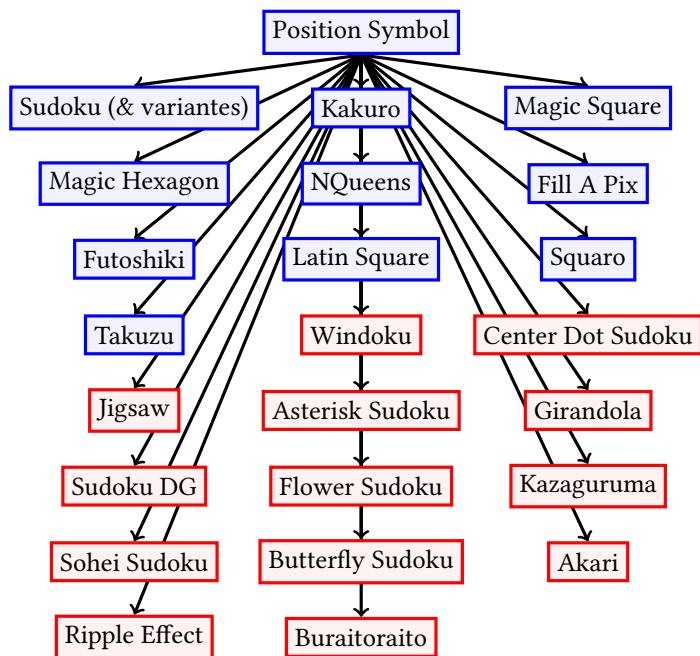


Figure 6.1: Position Symbol Category

- New Path category puzzles with Loop are also available, including Big Tour and Masyu (fig:6.2).

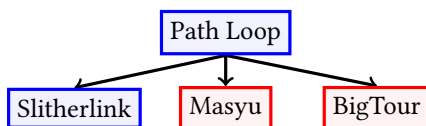


Figure 6.2: Path Loop Category

The new categories include :

- Shading Binary Contiguous puzzles with, for example Hitori, Nonogram or Tilepaint (fig: 6.3).

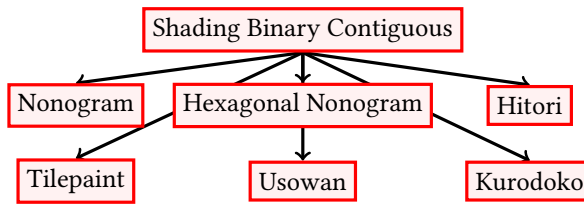


Figure 6.3: Shading Binary Contiguous Category

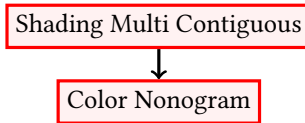


Figure 6.4: Shading Multi Contiguous Category

- Shading Multi Contiguous puzzles with the Color Nonogram (fig: 6.4).
- Path Connection-style puzzles with Hashiwokakero (fig: 6.5).

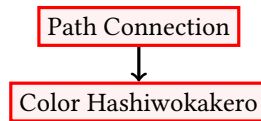


Figure 6.5: Path Connection Category

Despite all the additions, many other puzzles in these five categories can still be added (Yajilin⁵ or Suraromu⁶, for example). To do this, you will need to add new conditions and mechanics to the puzzles. For example, it is now possible to have several hints that also use an attribute to determine a colour. Adding a new attribute to the hint object would make it possible to add a direction to the hints.

In addition, Path Vector and Position Shape puzzles are not yet covered. In the implementation paths, a direction method should be implemented in order to implement vectors as playable pieces. For Position Shape puzzles, a geometric shape detection method would greatly help their implementation.

These different themes have not been implemented, mainly due to a lack of time. Implementing new ludemes like IsMatch, for example, sometimes takes a lot of time. Changes sometimes have to be made to the system and, above all, each implementation has to be efficient and optimised.

⁵<https://www.nikoli.co.jp/en/puzzles/yajilin/>

⁶<https://www.nikoli.co.jp/en/puzzles/suraromu/>

6.2.2 CREATING AI TO SOLVE PUZZLES

Ludii has a built-in AI system. For example, it is possible to test AIs on various games such as chess or Go. At present, the AIs in the system are not trained to solve deduction puzzles. The development of solving elements using deep reinforcement learning can prove interesting. As shown in [1], it uses two learning techniques to solve Sudoku. They use Convolutional Neural Networks and Bidirectional LSTMs. In their study, they demonstrate the effectiveness of this approach in solving a puzzle such as Sudoku. The approach using Convolutional Neural Networks produced the best results. This theme could be extended to several puzzles in order to solve them. Learning techniques could therefore compensate for the fact that search algorithms are inadequate. Another approach in the same theme would be to combine search algorithms and Deep Learning algorithms. As shown in [19], it is possible to solve the Jigsaw puzzle (a variant of Sudoku) using the Monte Carlo search algorithm. In this study, this algorithm is trained using a deep learning method that also uses the Neural Network concept. This is why it is important that the puzzles are compatible with the design of AI agents, as this would enable new studies to be carried out in Ludii.

6.2.3 GENERATING NEW INSTANCES

At the moment, to add new instances, we have to create them ourselves, and we don't yet have a way of automatically checking that our new instance is functional and leads to a solution. Once the solver is up and running, we could imagine a puzzle generation system that would generate new instances. This system would take the information from the puzzle, create an instance of the puzzle and check that it is correct. If it is, the system would remove certain information from the puzzle and offer it to the players. In this way, Ludii could offer a multitude of Sudoku grids or other puzzles to players.

In [32], the authors describe the concept of Procedural Content Generation (PCG). This corresponds to the creation of game content using algorithmic means. It would be interesting to take this concept and apply it to puzzles. In this way, puzzle generation would propose a complete instance with, for example, new regions, new hints, etc. The puzzle would be proposed on condition that it had at least one solution. In the same vein, LLMs (Large Language Models)⁷ could be used. An LLM will take an input and generate elements according to probability. This is defined by training the LLM. It will propose an appropriate response to the request using the elements generated. The training must therefore be appropriate to the different puzzle concepts.

If this generation sees the light of day, it could have an impact on the industrial world. It would make it easy to have a very large number of instances of a puzzle. What's more, it could help companies like Nikoli who, even today, create each instance manually.

⁷https://en.wikipedia.org/wiki/Large_language_model

7

CONCLUSION

To conclude this thesis, the approach has enabled us to model twenty-three new puzzles and add ten new ludemes (new mechanics). According to the taxonomy[15], the two existing categories are composed of new puzzles and three new categories are now available in the Ludii system. In addition to the new puzzles and new mechanics, new graphic styles have been introduced to ensure that each puzzle is modelled as closely as possible to its paper version. All the deduction puzzles in Ludii (old and new) can be found in Appendix C (7.3).

These various additions had to be tested to ensure that each new ludemes was effective and that the edemic representations of the puzzles were as clear as possible. As shown by the first two experiments, the use of each gameplay element is rapid, as each puzzle can perform a large number of moves in forty seconds. When the instances are larger, the number of moves performed decreases, but this is due to the pre-calculation in the system itself and is not linked to the game. The only puzzle with a different obsevation is the Nonogram. The ludeme in this puzzle performs too much calculation (because it generates cases in a combinatorial way). This is therefore an area for improvement. In the second experiment, all the puzzles had a fairly short description, which made the puzzle easy to understand. The most extensive ludemic representations come from the need to define a large number of elements, such as regions, in the puzzle.

All these contributions should be available in the next release of the Ludii system. This thesis will therefore have enabled us to make a contribution to an international open-source project.

Now that the puzzles can be modelled for the most part, others are not yet. New mechanics will have to be added to the system in order to be able to model them. Now that the modelling of different categories of puzzles is possible, there is a solving section to be done. This will be done by converting them to XCSP so that they can be solved using different solvers. This section is currently being developed in another thesis: General Game Playing and Constraint Programming to Solve any Logic Puzzles.

BIBLIOGRAPHY

REFERENCES

- [1] Akin-David, C. and Mantey, R. (2018). Solving sudoku with neural networks.
- [2] Apt, K. (2003). *Principles of constraint programming*. Cambridge university press.
- [3] Batenburg, K. J. and Kusters, W. A. (2009). Solving nonograms by combining relaxations. *Pattern Recognition*, 42(8):1672–1683.
- [4] Beck, K. (2022). *Test driven development: By example*. Addison-Wesley Professional.
- [5] Berend, D., Pomeranz, D., Rabani, R., and Raziel, B. (2014). Nonograms: Combinatorial questions and algorithms. *Discrete Applied Mathematics*, 169:30–42.
- [6] Boussemart, F., Lecoutre, C., Audemard, G., and Piette, C. (2016). Xcsp3: an integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398*.
- [7] Browne, C. (2013). Deductive search for logic puzzles. In *2013 IEEE Conference on Computational Intelligence in Games (CIG)*, pages 1–8. IEEE.
- [8] Browne, C. (2016). A class grammar for general games. In *International Conference on Computers and Games*, pages 167–182. Springer.
- [9] Browne, C., Piette, É., Stephenson, M., and Soemers, D. J. (2021). General board geometry. In *Advances in Computer Games*, pages 235–246. Springer.
- [10] Browne, C., Soemers, D., Piette, E., Stephenson, M., and Crist, W. (2020). *Ludii language reference*. PhD thesis, Maastricht University.
- [11] Browne, C., Soemers, D. J., Piette, É., Stephenson, M., Conrad, M., Crist, W., Depaulis, T., Duggan, E., Horn, F., Kelk, S., et al. (2019). Foundations of digital arch {\ae} oludology. *arXiv preprint arXiv:1905.13516*.
- [12] Browne, C. B. (2008). *Automatic generation and evaluation of recombination games*. PhD thesis, Queensland University of Technology.
- [13] Crawford, B., Castro, C., and Monfroy, E. (2009). *Solving Sudoku with Constraint Programming*, volume 35, pages 345–348.
- [14] Crist, W., Piette, E., Soemers, D. J., Stephenson, M., and Browne, C. (2022). Computational approaches for recognising and reconstructing ancient games: The case of ludus latrunculorum.

- [15] Hufkens, L. V. and Browne, C. (2019). A functional taxonomy of logic puzzles. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE.
- [16] Kowalski, J., Mika, M., Sutowicz, J., and Szykuła, M. (2019). Regular boardgames. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1699–1706.
- [17] Liu, C., Huang, S., Naying, G., Khalid, M. N. A., and Iida, H. (2022). A solver of single-agent stochastic puzzle: A case study with minesweeper. *Knowledge-Based Systems*, 246:108630.
- [18] Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M. (2008). General game playing: Game description language specification.
- [19] Paumard, M.-M., Tabia, H., and Picard, D. (2023). Alphazzzle: Jigsaw puzzle solver with deep monte-carlo tree search. *arXiv preprint arXiv:2302.00384*.
- [20] Piette, C., Piette, E., Stephenson, M., Soemers, D. J., and Browne, C. (2019a). Ludii and xcsp: playing and solving logic puzzles. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE.
- [21] Piette, É., Browne, C., and Soemers, D. J. (2021a). Ludii game logic guide. *arXiv preprint arXiv:2101.02120*.
- [22] Piette, É., Crist, W., Soemers, D. J., Rougetet, L., Courts, S., Penn, T., and Morenville, A. (2024). Gametable cost action: kickoff report. *ICGA Journal*, (Preprint):1–17.
- [23] Piette, E., Rougetet, L., Crist, W., Stephenson, M., Soemers, D., and Browne, C. (2021b). A ludii analysis of the french military game. In *XXIII Board Game Studies*.
- [24] Piette, E., Soemers, D. J., Stephenson, M., Sironi, C. F., Winands, M. H., and Browne, C. (2019b). Ludii—the ludemic general game system. *arXiv preprint arXiv:1905.05013*.
- [25] Piette, E., Stephenson, M., Soemers, D. J., and Browne, C. (2019c). An empirical evaluation of two general game systems: Ludii and rbg. In *2019 IEEE Conference on Games (CoG)*, pages 1–4. IEEE.
- [26] Pitrat, J. (1968). Realization of a general game-playing program. In *IFIP congress (2)*, pages 1570–1574.
- [27] Sedgewick, R. and Wayne, K. (2011). *Algorithms*, forth edition.
- [28] Simonis, H. (2008). Kakuro as a constraint problem. *Proc. seventh Int. Works. on Constraint Modelling and Reformulation*.
- [29] Soemers, D., Piette, É., Stephenson, M., and Browne, C. (2022). Ludii user guide. Technical report, Technical report, Maastricht University. [https://ludii.games/downloads ...](https://ludii.games/downloads...)
- [30] Soemers, D. J., Piette, É., Stephenson, M., and Browne, C. (2021). Optimised playout implementations for the ludii general game system. In *Advances in Computer Games*, pages 223–234. Springer.

- [31] Stephenson, M., Piette, E., Soemers, D. J., and Browne, C. (2019). An overview of the ludii general game system. In *2019 IEEE Conference on Games (CoG)*, pages 1–2. IEEE.
- [32] Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. (2011). Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186.

APPENDIX

7.1 APPENDIX A : RESULT EXPERIEMENT SIZE OF CHALLENGE

Name And Size	Execution Time (s)	$N^{br}OfMovements$	$AverageN^{br}OfMovements$
Tridoku	40	10,930	273.23
Magic Square 3x3	40	248,816	6,085.82
Magic Square 4x4	40	47,859	1,167.61
Magic Square 5x5	40.001	13,749	342.70
Magic Square 6x6	40.003	3,385	90.72
Magic Square 7x7	40.02	1,430	38.52
Magic Square 8x8	40.03	672	18.20
Magic Square 9x9	40.01	340	9.21
Magic Square 10x10	40.02	186	5.07
Magic Square 11x11	40.20	107	2.92
Magic Square 12x12	40.34	64	1.73
Magic Square 13x13	40.37	40	1.08
Magic Square 14x14	41.22	26	0.68
Magic Square 15x15	40.56	17	0.46
Hitori 6x6	40	42,105	1,053.28
Hitori 10x10	40	5,070	122.32
Hitori 8x8	40	12,954	316.24
Hitori 5x5	40	88,243	2,194.97
Hitori 10x10	40	5,053	122.39
Hitori 15x15	40.02	908	21.63
Hitori 20x20	40.15	246	6.12
Takuzu 10x10	40	33,589	824.51
Takuzu 12x12	40	19,437	482.26
Takuzu 14x14	40	7,643	186.53
Takuzu 20x20	40	2,382	58.42
Kakuro 4x4	40	247,779	5,997.17
Kakuro 6x6	40	54,391	1,323.77
Kakuro 11x11	40	7,384	182.31
Kakuro 17x17	40.03	1,287	31.89
Kakuro 31x31	40.19	130	3.21
Kakuro 14x12	40	3,628	89.61
Kakuro 12x10	40	6,713	165.71

Name And Size	Execution Time (s)	N^{br} Of Movements	Average N^{br} Of Movements
Sudoku DG 9x9	40	6,418	160.37
Sudoku DG 9x9	40	6,490	162.27
Nonogram 5x5	40	75,503	1903.003
Nonogram 10x10	40.03	573	14.83
Nonogram 15x15	44.1	7	0.16
Nonogram 20x20	1,177.35	1	8.49E-4
Sudoku Mine 9x9	40	7,649	191.22
Girandola 9x9	40	7,566	189.03
Girandola 9x9	40	7,179	179.24
Fill A Pix 9x9	40	4,243	106.05
Kurodoko 7x7	40	21,788	532.93
Kurodoko 9x9	40	5,063	121.03
Kurodoko 9x9	40	5,324	130.96
Jigsaw 9x9	40	9,626	241.27
Jigsaw 9x9	40	9,107	228.12
Slitherlink 5x5	40	80,193	2,004.11
Slitherlink 7x7	40	20,178	503.37
Slitherlink 8x8	40	13,064	327.93
Slitherlink 11x11	40	3,158	78.55
Slitherlink 16x16	40.06	571	14.28
Slitherlink 21x21	40.1	160	3.96
Slitherlink 31x26	40.02	35	0.85
Slitherlink 31x46	40.36	8	0.15
Butterfly Sudoku	40	4,123	103.07
Killer Sudoku	40	3,185	79.62
NQueens 4x4	40	263,037	6,522.68
NQueens 5x5	40	94,595	2,347.83
NQueens 6x6	40	40,149	988.94
NQueens 7x7	40	18,250	447.65
NQueens 8x8	40	8,992	218.98
NQueens 9x9	40	4,783	115.59
NQueens 10x10	40	2,716	64.13
NQueens 11x11	40	1,596	37.47
NQueens 12x12	40	957	22.55
NQueens 13x13	40.01	599	14.22
NQueens 14x14	40.01	385	9.22
NQueens 15x15	40.03	257	6.17
NQueens 16x16	40.04	176	4.22
NQueens 17x17	40.05	122	2.92
NQueens 18x18	40.05	88	2.12
NQueens 19x19	40.48	64	1.58
NQueens 20x20	40.62	49	1.18
NQueens 21x21	40.59	37	0.88
NQueens 22x22	40.79	28	0.68
NQueens 23x23	41.69	22	0.52
NQueens 24x24	41.36	17	0.41
NQueens 25x25	42.83	14	0.32

Name And Size	Execution Time (s)	N^{br} Of Movements	Average N^{br} Of Movements
NQueens 26x26	42.07	11	0.26
NQueens 27x27	43.3	9	0.21
NQueens 28x28	42.02	7	0.16
NQueens 29x29	43.32	6	0.14
NQueens 30x30	44.99	5	0.11
Akari 7x7	40	44,157	1,113.79
Akari 7x7	40	39,646	1,008.87
Akari 14x14	40	1,984	50.45
Akari 25x25	40.04	101	2.4
Kazaguruma	40.04	988	24.68
Asterisk Sudoku	40	10,246	252.88
Asterisk Sudoku	40	9,068	225.57
Samurai Sudoku	40	774	19.35
Hashiwokakero 17	40	222,952	5,560.72
Hashiwokakero 25	40	111,585	2,796.31
Hashiwokakero 33	40	67,637	1,700.64
Hashiwokakero 89	40	7,831	196.64
Big Tour 10x10	40	18,319	457.95
Hexagonal Nonogram 3x3	40	104,105	2,602.60
Color Nonogram 5x5	40.03	872	21.78
Sudoku X 9x9	40	8,260	206.47
Windoku 9x9	40	9,485	233.81
Windoku 9x9	40	11,416	282.93
Masyu 7x7	40	28,804	716.12
Masyu 8x8	40	17,193	429.34
Masyu 10x10	40	5,477	135.45
Flower Sudoku	40.01	2,064	51.58
Magic Hexagon	40	27,063	676.56
Latin Square 2x2	40	2,675,284	66,802.16
Latin Square 3x3	40	752,986	18,659.16
Latin Square 4x4	40	259,712	6,437.39
Latin Square 5x5	40	110,755	2,768.80
Latin Square 6x6	40	55,210	1,371.62
Latin Square 7x7	40	29,439	730.13
Latin Square 8x8	40	13,469	332.61
Latin Square 9x9	40	7,979	198.41
Latin Square 10x10	40.01	4,925	122.56
Latin Square 11x11	40.01	3,197	79.65
Latin Square 12x12	40.01	2,126	52.98
Latin Square 13x13	40.01	1,445	35.95
Latin Square 14x14	40.03	1,017	25.22
Latin Square 15x15	40	729	18.19
Latin Square 16x16	40.03	377	9.37
Latin Square 17x17	40.02	280	6.98
Latin Square 18x18	40.11	209	5.19
Latin Square 19x19	40.16	160	4.02
Latin Square 20x20	40.25	123	3.03

Name And Size	Execution Time (s)	$N^{br}OfMovements$	Average $N^{br}OfMovements$
Latin Square 21x21	40.30	97	2.45
Latin Square 22x22	40.24	79	1.95
Latin Square 23x23	40.63	64	1.57
Latin Square 24x24	40.43	52	1.27
Latin Square 25x25	40.40	42	1.03
Latin Square 26x26	40.76	35	0.85
Latin Square 27x27	40.75	29	0.70
Latin Square 28x28	40.28	24	0.59
Latin Square 29x29	40.17	24	0.49
Latin Square 30x30	40.49	17	0.42
Buraitoraito	40	12,258	306.45
Squaro	40	115,408	2,885.19
Sujiken	40	33,443	836.06
Anti-Knight Sudoku	40	1,349	33.72
RippleEffect 7x7	40	18,440	463.49
RippleEffect 7x7	40	15,975	402.22
RippleEffect 6x6	40	31,225	792.36
RippleEffect 10x10	40	4,244	107.44
RippleEffect 17x17	40.08	328	8.23
Hoshi 17x17	40.08	27,285	682.10
Tilepaint 5x5	40	1,055,809	26,132.17
Tilepaint 11x11	40	90,332	2,242.68
Tilepaint 11x11	40	88,527	2,209.35
Tilepaint 16x16	40	23,370	584.26
Sohei Sudoku	40.02	1,387	34.65
Center Dot Sudoku 9x9	40.02	11,050	273.95
Center Dot Sudoku 9x9	40.02	7,839	194.26
Usowan	40	275,256	6,881.4
Futoshiki	40	80,932	1,943.35
Futoshiki	40	71,503	1,753.47
Sudoku 9x9	40	8,694	217.37
Sudoku 9x9	40	10,848	269.10

7.2 APPENDIX B : RESULT EXPERIMENT THE NUMBER OF TOKEN

Name	$N^{br}OfTokens$
Tridoku	267
Magic Square	36
Hitori	141
Takuzu	82
Kakuro	61
Sudoku DG	184
Nonogram	126
Sudoku Mine	105
Girandola	99
Fill A Pix	157
Kurodoko	83
Jigsaw	185
Slitherlink	73
Butterfly Sudoku	1059
Killer Sudoku	178
N Queens	36
Akari	210
Kazaguruma	1602
Color Nonogram	150
Asterisk Sudoku	100
Samurai Sudoku	1113
Hashiwokakero	180
Big Tour	92
Hexagonal Nonogram	154
Sudoku X	95
Windoku	132
Masyu	101
Flower Sudoku	1007
Magic Hexagon	104
Latin Square	30
Buraitoraito	153
Squaro	123
Sujiken	268
Anti-Knight Sudoku	108
RippleEffect	132
Hoshi	336
Tilepaint	113
Sohei Sudoku	1402
Center Dot Sudoku	105
Usowan	105
Futoshiki	65
Sudoku	74

7.3 APPENDIX C : ALL THE DEDUCTION PUZZLES IN LUDII

3	9	1	5	2	7	6	8	4
4	2	5	6	8	9	7	3	1
8	7	6	4	3	1	9	5	2
7	8	9	3	1	2	4	6	5
6	5	3	9	7	4	2	1	8
2	1	4	8	6	5	3	7	9
1	4	7	2	5	3	8	9	6
5	6	2	7	9	8	1	4	3
9	3	8	1	4	6	5	2	7

(a) Antiknight
Sudoku

2	1	5	6	4	7	3	9	8
3	6	8	9	5	2	1	7	4
7	9	4	3	8	1	6	5	2
5	8	6	2	7	4	9	3	1
1	4	2	5	9	3	8	6	7
9	7	3	8	1	6	4	2	5
8	2	1	7	3	9	5	4	6
6	5	9	4	2	8	7	1	3
4	3	7	1	6	5	2	8	9

(b) Killer Sudoku

3	5	1	6	8	7	9	2	4
8	4	7	9	5	2	3	1	6
2	9	6	3	1	4	7	8	5
5	6	4	1	2	9	8	7	3
7	8	2	4	3	5	6	9	1
1	3	9	8	7	6	5	4	2
4	7	3	2	6	8	1	5	9
9	1	5	7	4	3	2	6	8
6	2	8	5	9	1	4	3	7

(c) Samurai
Sudoku

1	1		1					

(d) Sudoku Mine

6	3	9	8	4	1	2	7	5
7	2	4	9	5	3	1	6	8
1	8	5	7	2	6	3	9	4
2	5	6	1	3	7	4	8	9
4	9	1	5	8	2	6	3	7
8	7	3	4	6	9	5	2	1
5	4	2	3	9	8	7	1	6
3	1	8	6	7	5	9	4	2
9	6	7	2	1	4	8	5	3

(e) Sudoku X

5								
6	3	7						
2	8	9	4	1				
3	5	7	3	2	6			
4	8	2	1	6	9	7	5	3
1	6	9	7	5	4	1	4	9
7	5	3	2	1	9	8	5	3
8	2	1	6	7	8	5	3	2
9	6	5	4	3	2	1	9	7

(f) Tridoku

8								
3	4							
1	2	9						
4	8	5	6					
7	6	2	9	5				
9	3	1	4	8	7			
5	7	8	3	6	1	2		
2	1	4	5	7	9	6	3	
6	9	3	2	4	8	5	7	1

(g) Hoshi

8								
3	4							
1	2	9						
4	8	5	6					
7	6	2	9	5				
9	3	1	4	8	7			
5	7	8	3	6	1	2		
2	1	4	5	7	9	6	3	
6	9	3	2	4	8	5	7	1

(h) Sujiken

8	1	2	7	5	3	6	4	9
9	4	3	6	8	2	1	7	5
6	7	5	4	9	1	2	8	3
1	5	4	2	3	7	8	9	6
3	6	9	8	4	5	7	2	1
2	8	7	1	6	9	5	3	4
5	2	1	9	7	4	3	6	8
4	3	8	5	2	6	9	1	7
7	9	6	3	1	8	4	5	2

(i) Sudoku

5	2	3	7	5	1	5
3	1	6	9	7	3	4
5	7	7	8	9	5	2
3	1	5	9	3	8	7
1	2	3	4	1	2	4
2	4	5	9	7	4	8
9	8	7	5	5	7	9
9	8	7	5	5	7	9
1	3	2	4	3	9	7
2	1	1	3	8	9	2
6	8	9	5	6	9	5
7	9	8	9	7	6	7

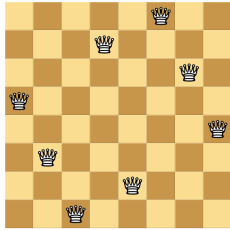
(j) Kakuro

2	7	6
9	5	1
4	3	8

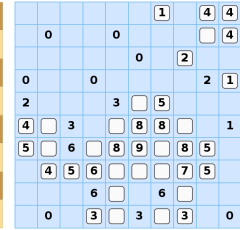
(k) Magic Square

3	1	7	1	8
1	9	7	1	1
1	6	2	5	6
1	2	4	8	1
1	0	1	3	1

(l) Magic Hexagon



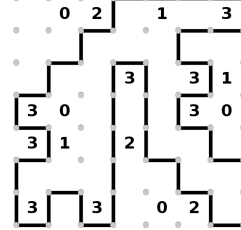
(a) NQueens



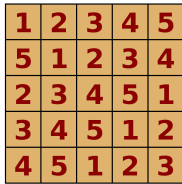
(b) Fill A Pix



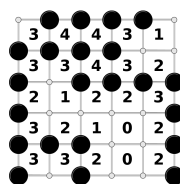
(c) Futoshi



(d) Slitherlink



(e) Latin Square



(f) Squaro



(g) Takuzu



(h) Center Dot Sudoku



(i) Windoku



(j) Jigsaw



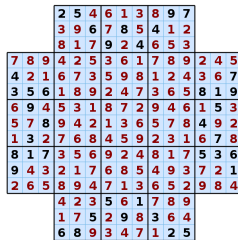
(k) Asterisk Sudoku



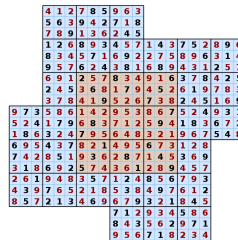
(l) Girandola



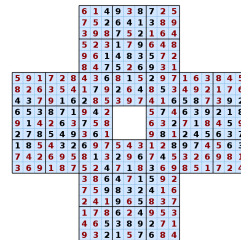
(m) Sudoku DG



(n) Flower Sudoku



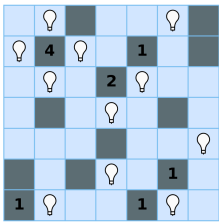
(o) Kazaguruma



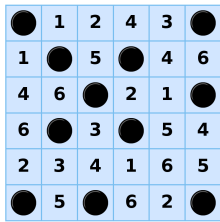
(p) Sohei Sudoku



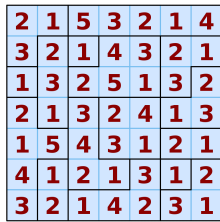
(a) Butterfly Sudoku



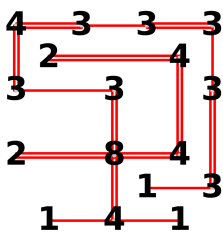
(b) Akari



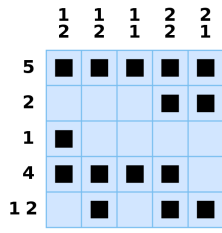
(c) Hitori



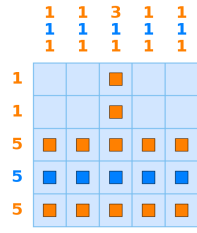
(d) Ripple Effect



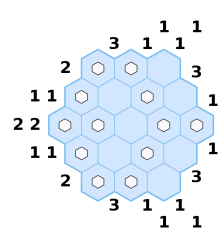
(e) Hashiwokakero



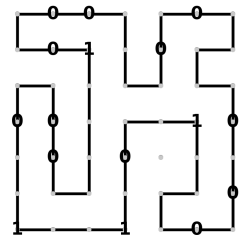
(f) Nonogram



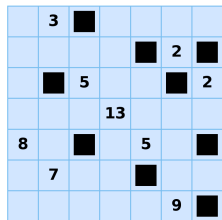
(g) Color Nonogram



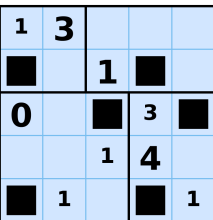
(h) Hexagonal Nonogram



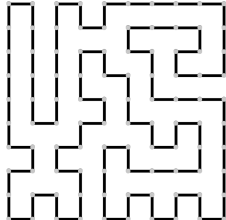
(i) Masyu



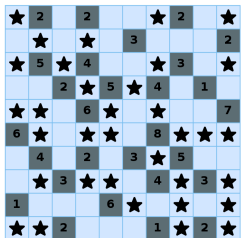
(j) Kurodoko



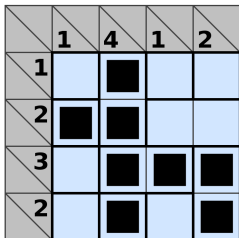
(k) Usowan



(l) Big Tour



(m) Buraitoraito



(n) Tilepaint

