Planning Sudoku

Planning Sudoku for Hierarchical Domain Definition Language

1st Bruno Campos Ribeiro
Faculdade do Gama
Universidade de Brasília)
Brasília, Brasil
bbrunoo@icloud.com

2nd Igor e Silva Penha
Faculdade do Gama
Universidade de Brasília
Brasília, Brasil
igor.penharol@gmail.com

3rd Lucas Gobbi Bergholz
Faculdade Gama
Universidade de Brasília
Brasília, Brasil
lucas.bergholz@gmail.com

4th Danilo Carvalho Antunes
Faculdade Gama
Universidade de Brasília
Brasília, Brasil
danilocarvalhoantunes@gmail.com

ABSTRACT

In this study, we tackle the intricate logic-based game: Sudoku. The goal of Sudoku is to fill a 9x9 grid with numbers in such a way that each row, column, and each of the nine 3x3 subgrids that compose the grid contain all of the digits from 1 to 9. Ensuring player retention and satisfaction is paramount; thus, it is beneficial to prevent the discovery of trivial solutions that undermine the game's complexity intended by its creators. Consequently, the availability of tools that can verify the optimality of solutions is of great importance.

Sudoku is typically approached as a logic puzzle, but it can also be framed as a planning problem. In this work, we propose modelling Sudoku in Hierarchical Domain Definition Language (HDDL), which is an extension of PDDL supporting hierarchical planning. We explore how HDDL's structured representation of planning problems can effectively capture the constraints and rules of Sudoku.

Our investigation reveals that while HDDL allows for a structured and understandable model, converting the problem directly into a Satisfiability Problem (SAT) can yield more efficient solving. This efficiency is largely due to the ease of expressing the constraints of Sudoku in the language of SAT, which facilitates the discovery of concise and optimal solutions. In contrast, employing HDDL's expressive power to model such constraints does not significantly expedite the solution process when utilizing contemporary planning tools.

I. INTRODUCTION

Introduction:

The captivating allure of Sudoku, a logic-based puzzle game, has intrigued and engaged minds worldwide since its inception. Originally called "Number Place", Sudoku was popularized in Japan in the 1980s and has since evolved into a global phenomenon known for its simple rules yet profound depth of logic. The essence of Sudoku lies in filling a 9x9 grid with digits from 1 to 9 in a manner that each row, column, and 3x3 subgrid contains all these digits without repetition. This game, transcending mere entertainment, has emerged as a vibrant field of academic and computational research, particularly in the realms of logic, optimization, and problem-solving.

Parallel to the evolution of Sudoku, the Hierarchical Domain Definition Language (HDDL), an extension of the Planning Domain Definition Language (PDDL), has made significant strides. HDDL, introduced to support hierarchical planning in complex scenarios, has expanded the horizons of computational problem-solving. It allows for a structured approach to decomposing intricate problems into manageable sub-tasks, fostering more efficient solution strategies. The development of HDDL has been a testament to the progress in artificial intelligence and computational logic, catering to the growing need for solving complex, hierarchical problems in various domains.

The connection between Sudoku and HDDL is not immediately apparent, yet it is profoundly symbiotic. While Sudoku represents a paradigm of logical puzzles demanding strategic planning and problem decomposition, HDDL offers the computational framework to model and solve such problems. By framing Sudoku as a planning problem, this study explores the application of HDDL to Sudoku, leveraging its hierarchical planning capabilities to model the game's constraints and rules systematically. This approach not only provides a fresh perspective on solving Sudoku but also exemplifies the versatility and applicability of HDDL in translating a real-world logic problem into a structured computational model.

Furthermore, this study delves into the efficiency of solving Sudoku by directly converting it into a Satisfiability Problem (SAT). The SAT approach benefits from the straightforward expression of Sudoku's constraints in its language, facilitating the discovery of solutions that are both concise and optimal. This comparison between HDDL and SAT in the context of Sudoku not only illuminates the strengths and limitations of these methodologies but also contributes to the broader discourse on the most effective computational techniques for problem-solving in logic-based games.

In sum, this article embarks on a journey through the computational landscapes of Sudoku and HDDL, unraveling the intricate connection between a beloved puzzle and an advanced computational language. It aims to offer insights into how traditional games like Sudoku can be transformed through modern computational approaches, highlighting the

Identify applicable funding agency here. If none, delete this.

advancements in problem-solving techniques and their potential applications in various domains.

II. The Game

Sudoku, popular form of number game. In its simplest and most common configuration, sudoku consists of a 9 × 9 grid with numbers appearing in some of the squares. The object of the puzzle is to fill the remaining squares, using all the numbers 1–9 exactly once in each row, column, and the nine 3 × 3 subgrids

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

Fig. 1. Sudoku Board Example.

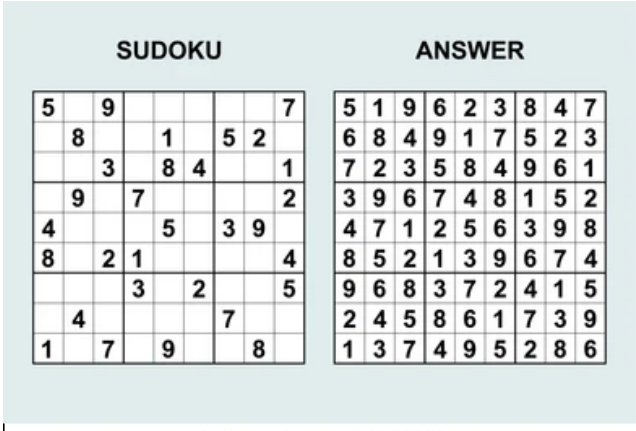

Fig. 2. Sudoku Solved.

III. HDDL Formulation

Given the well-mapped nature of Sudoku as a problem, object instantiations are not necessary; all objects are constants. The considered objects fall into the following types: row and col, both of which are sub-types of position, box and digit.

In this paradigm, the types 'row' and 'col' are representative of discrete positions on the Sudoku grid, delineating subcategories beneath the more encompassing type, 'position.' Concurrently, 'box' and 'digit' serve as distinct types contributing to the puzzle's structural composition.

This methodological approach streamlines the representation of Sudoku, accentuating the immutable nature of its constituent elements. In contrast to scenarios featuring dynamic objects necessitating instantiation, the static attributes of Sudoku permit the treatment of its components as constants.

Listing 1. Sudoku Domain Definition in HDDL

```
(define (domain sudoku)
(:types row col - position digit box)
(:constants
    d1 d2 d3 d4 d5 d6 d7 d8 d9 - digit
    ↪
    b0 b1 b2 b3 b4 b5 b6 b7 b8 - box
    c0 c1 c2 c3 c4 c5 c6 c7 c8 - col
    r0 r1 r2 r3 r4 r5 r6 r7 r8 - row)
)
```

'position': Represents the general concept of a position within the Sudoku grid.

Relationship: The ordered pair ('row' and 'col') is a sub-type of 'position,' emphasising the unique spatial identification of a cell.

'row' and 'col' as an Ordered Pair Position: Together, 'row' and 'col' form an ordered pair, representing a specific position on the Sudoku grid.

Relationship: The combination of 'row' and 'col' acts as a coordinated pair within the Sudoku grid, uniquely identifying a cell's position. This ordered pair is a sub-type of 'position,' highlighting its role as a spatial element.

'box': Represents the 3x3 boxes or regions in the Sudoku grid.

Relationship: 'box' remains a distinct type, encapsulating the grouping of cells into non-overlapping 3x3 regions. The ordered pair ('row' and 'col') and 'box' collectively contribute to the structural composition of the puzzle.

'digit': Represents the digits (numbers 1 to 9) used in the Sudoku puzzle.

Relationship: 'digit' continues to be crucial for expressing the numerical aspect of Sudoku, representing the possible values that can fill each cell.

From delineating the spatial coordinates of cells within designated boxes, columns, and rows to capturing the manifestation of digits and the occupied status of cells, these predicates form integral components of the Sudoku code. Join us as we meticulously dissect the semantics underlying predicates such as cell-at-box, digit-at, digit-at-box, filled and inc.

Listing 2. Sudoku Domain Definition in HDDL

```
(:predicates
    (cell-at-box ?b - box ?c - col ?r -
    ↪ row)
    (digit-at ?d - digit ?c - col ?r -
    ↪ row)
    (digit-at-box ?d - digit ?b - box)
    (filled ?c - col ?r - row)
    (inc ?a ?b - position))
```

(cell-at-box ?b - box ?c - col ?r - row): This predicate asserts the existence of a cell at the intersection of a specified box (?b), column (?c), and row (?r). It serves to pinpoint a precise spatial location within the Sudoku grid, contributing to the overall structure of the puzzle.

(digit-at ?d - digit ?c - col ?r - row): The digit-at predicate signifies the presence of a particular digit (?d) at a specified column (?c) and row (?r). This encapsulates the numerical aspect of Sudoku, representing the values assigned to specific cells and forming a crucial part of the puzzle's logical constraints.

(digit-at-box ?d - digit ?b - box): Denoting the coexistence of a digit (?d) within a designated box (?b), this predicate enforces constraints related to the distribution of digits within specific regions of the Sudoku grid. It contributes to the overall logic governing the arrangement of numbers in the puzzle.

(filled ?c - col ?r - row): The filled predicate indicates that a specific cell at the intersection of a given column (?c) and row (?r) is occupied or filled. It captures the state where a cell is no longer empty, a critical condition in Sudoku puzzle-solving and a key element in maintaining puzzle integrity.

(inc ?a ?b - position): The inc predicate introduces an incremental relationship between two positions (?a and ?b). It implies a sequential or ordered connection, contributing to the logical constraints inherent in Sudoku. This predicate helps capture the sequential arrangement of positions within the puzzle grid.

In the realm of Sudoku modelling, we unravel the intricacies of tasks that form the very foundation of puzzle-solving strategies. Within this complex terrain, tasks like 'all-check,' 'fill-cell,' 'check-box,' 'check-row,' and 'check-column' stand out as pivotal components.

Listing 3. Sudoku Domain Definition in HDDL

```
(:task all-check :parameters())
(:task fill-cell :parameters
    (?b - box ?c - col ?r - row ?d -
      ↪ digit))
(:task check-box :parameters
    (?b - box ?d - digit))
(:task check-row :parameters
    (?r - row ?d - digit))
(:task check-column :parameters
    (?c - col ?d - digit))
```

(:task all-check :parameters()): The 'all-check' task takes a panoramic view without specific parameters, offering a comprehensive analysis that transcends the boundaries of individual puzzle elements. It serves as a meta-task, shedding light on the holistic puzzle landscape.

(:task fill-cell :parameters (?b - box ?c - col ?r - row ?d - digit)): In the 'fill-cell' task, precision meets action. Parameterized by a box (?b), column (?c), row (?r), and digit (?d), it encapsulates the strategic act of populating a specific cell with a designated digit, a pivotal move in the Sudoku solving choreography.

(:task check-box :parameters (?b - box ?d - digit)): The 'check-box' task, with parameters specifying a box (?b) and a digit (?d), directs our focus to a targeted examination within a specific box. Here, we probe the presence or absence of a particular digit, adding depth to the logical coherence of the puzzle.

(:task check-row :parameters (?r - row ?d - digit)): In the 'check-row' task, a nuanced lens zooms into a designated row (?r) and digit (?d), meticulously scrutinizing the occurrences of the digit along the horizontal axis. This task stands sentinel, ensuring the harmonious alignment of digits within rows.

(:task check-column :parameters (?c - col ?d - digit)): The 'check-column' task, with parameters specifying a column (?c) and a digit (?d), engages in a detailed exploration of a specific digit within the designated column. It plays a pivotal role in upholding the vertical coherence of digits, a cornerstone of Sudoku logic.

The methods play a crucial role in decomposing complex tasks into more manageable sub-tasks, allowing for a structured and hierarchical approach to problem-solving. Here's the methods which were used to decomposing our tasks:

Listing 4. Sudoku Domain Definition in HDDL

```
(:method m-all-check
        :parameters ()
        :task (all-check)
        :precondition (and (forall (?r -
          ↪ row ?c - col) (filled ?c
          ↪ ?r))))
(:method m-fill-cell
        :parameters (?b - box ?c - col
          ↪ ?r - row ?d - digit)
        :task (fill-cell ?b ?c ?r ?d)
        :precondition (and (not (filled
          ↪ ?c ?r)))
        :ordered-subtasks (and
          ↪ (check-box ?b ?d)
          ↪ (check-row ?r ?d)
          ↪ (check-column ?c ?d)
          ↪ (FILL-CELL ?b ?c ?r ?d)))
(:method m-check-row
        :parameters (?c0 ?c1 ?c2 ?c3 ?c4
          ↪ ?c5 ?c6 ?c7 ?c8 - col ?r -
          ↪ row ?d - digit)
        :task (check-row ?r ?d)
        :precondition (and (inc ?c0 ?c1)
          ↪ (inc ?c1 ?c2) (inc ?c2
          ↪ ?c3) (inc ?c3 ?c4) (inc
          ↪ ?c4 ?c5) (inc ?c5 ?c6)
          ↪ (inc ?c6 ?c7) (inc ?c7 ?c8)
              (forall (?c - col) (not
                ↪ (digit-at ?d ?c
                ↪ ?r)))))
(:method m-check-column
        :parameters (?c - col ?r0 ?r1
          ↪ ?r2 ?r3 ?r4 ?r5 ?r6 ?r7
```

```
        ↪ ?r8 - row ?d - digit)
    :task (check-column ?c ?d)
    :precondition (and (inc ?r0 ?r1)
        ↪ (inc ?r1 ?r2) (inc ?r2
        ↪ ?r3) (inc ?r3 ?r4) (inc
        ↪ ?r4 ?r5) (inc ?r5 ?r6)
        ↪ (inc ?r6 ?r7) (inc ?r7 ?r8)
            (forall (?r - row) (not
                ↪ (digit-at ?d ?c
                ↪ ?r)))))
(:method m-check-box
    :parameters (?b - box ?d - digit)
    :task (check-box ?b ?d)
    :precondition (and (not
        ↪ (digit-at-box ?d ?b))))
```

m-all-check Method

- **Objective:** Ensure all cells in the Sudoku grid are filled.
- **Decomposition:** Divides the overarching goal into subgoals, checking each individual cell.
- **Preconditions:** Validates that each cell is filled.

m-fill-cell Method

- **Objective:** Fill a cell at a specified position with a digit.
- **Decomposition:** Hierarchically decomposes the task into checking the box, row, and column before filling the cell.
- **Preconditions:** Verifies that the cell is not already filled.

m-check-row and m-check-column Methods

- **Objective:** Check if a digit is valid in a row (column).
- **Decomposition:** Hierarchically decomposes the task into checking each individual cell within the row (column).
- **Preconditions:** Validates that the digit is not present in any other cell of the row (column).

m-check-box Method

- **Objective:** Validate if a digit is valid within a specific box.
- **Decomposition:** Simplifies the task by focusing on a specific box.
- **Preconditions:** Ensures the digit is not already present in the box.

The actions serve as executable steps that bring about changes in the state of the planning domain. Here's the only one action that were used to executable the Sudoku's domain:

Listing 5. Sudoku Domain Definition in HDDL

```
(:action FILL-CELL
    :parameters (?b - box ?c - col
        ↪ ?r - row ?d - digit)
    :precondition (and (cell-at-box
        ↪ ?b ?c ?r))
    :effect (and (digit-at ?d ?c ?r)
        ↪ (digit-at-box ?d ?b)
        ↪ (filled ?c ?r)))
```

Parameters:

- `?b - box`: Represents the box in which the cell is located.
- `?c - col`: Represents the column of the cell.
- `?r - row`: Represents the row of the cell.
- `?d - digit`: Represents the digit to be filled in the cell.

Precondition:

- `(cell-at-box ?b ?c ?r)`: Ensures that the specified cell is within the specified box.

Effect:

- `(digit-at ?d ?c ?r)`: Indicates that the specified digit is now present in the specified column and row.
- `(digit-at-box ?d ?b)`: Indicates that the specified digit is now present in the specified box.
- `(filled ?c ?r)`: Marks the specified cell as filled.

Listing 6. Sudoku Domain Definition in HDDL

```
(define (domain sudoku)
  (:requirements :hierarchy :typing :strips :universal-preconditions)
  (:types row col - position digit box)
  (:predicates
    (cell-at-box ?b - box ?c - col ?r - row)
    (digit-at ?d - digit ?c - col ?r - row)
    (digit-at-box ?d - digit ?b - box)
    (filled ?c - col ?r - row)
    (inc ?a ?b - position))
  (:constants
    d1 d2 d3 d4 d5 d6 d7 d8 d9 - digit
    b0 b1 b2 b3 b4 b5 b6 b7 b8 - box
    c0 c1 c2 c3 c4 c5 c6 c7 c8 - col
    r0 r1 r2 r3 r4 r5 r6 r7 r8 - row)
  (:task all-check :parameters())
  (:task fill-cell :parameters (?b - box ?c - col ?r - row ?d - digit))
  (:task check-box :parameters (?b - box ?d - digit))
  (:task check-row :parameters (?r - row ?d - digit))
  (:task check-column :parameters (?c - col ?d - digit))
  (:method m-all-check
    :parameters ()
    :task (all-check)
    :precondition (and (forall (?r - row ?c - col) (filled ?c ?r))))
  (:method m-fill-cell
    :parameters (?b - box ?c - col ?r - row ?d - digit)
    :task (fill-cell ?b ?c ?r ?d)
    :precondition (and (not (filled ?c ?r)))
    :ordered-subtasks (and (check-box ?b ?d) (check-row ?r ?d) (check-column ?c ?d)
      ↪ (FILL-CELL ?b ?c ?r ?d)))
  (:method m-check-row
    :parameters (?c0 ?c1 ?c2 ?c3 ?c4 ?c5 ?c6 ?c7 ?c8 - col ?r - row ?d - digit)
    :task (check-row ?r ?d)
    :precondition (and (inc ?c0 ?c1) (inc ?c1 ?c2) (inc ?c2 ?c3) (inc ?c3 ?c4) (inc
      ↪ ?c4 ?c5) (inc ?c5 ?c6) (inc ?c6 ?c7) (inc ?c7 ?c8)
      (forall (?c - col) (not (digit-at ?d ?c ?r)))))
  (:method m-check-column
    :parameters (?c - col ?r0 ?r1 ?r2 ?r3 ?r4 ?r5 ?r6 ?r7 ?r8 - row ?d - digit)
    :task (check-column ?c ?d)
    :precondition (and (inc ?r0 ?r1) (inc ?r1 ?r2) (inc ?r2 ?r3) (inc ?r3 ?r4) (inc
      ↪ ?r4 ?r5) (inc ?r5 ?r6) (inc ?r6 ?r7) (inc ?r7 ?r8)
      (forall (?r - row) (not (digit-at ?d ?c ?r)))))
  (:method m-check-box
    :parameters (?b - box ?d - digit)
    :task (check-box ?b ?d)
    :precondition (and (not (digit-at-box ?d ?b))))
  (:action FILL-CELL
    :parameters (?b - box ?c - col ?r - row ?d - digit)
    :precondition (and (cell-at-box ?b ?c ?r))
    :effect (and (digit-at ?d ?c ?r) (digit-at-box ?d ?b) (filled ?c ?r)))
```

IV. Benchmark

To choose which planners would be used to run the domain, the International Planning Competition (IPC) 2023 was used as a parameter, and the pandaPI and PandaDealer planners were chosen to complete this task because of the result PandaDealer achieved in this competition, winning three out of 6 tracks, such as Total-Order Agile and Total-Order Satisficing.

Unfortunatly, both of the planners were not able to run the domain. To correctly use both of them, it is necessary to follow a few steps: first, you need to parse the code, using the pandaPIparser. The second step is to convert this parsed code to ”sas” language, using pandaPIgrounder. The third and final step is to run the pandaPIengine with the sas code.

The problem with the sudoku domain happened at the second step, in which the pandaPIgrounder could not complete the transcription to sas language, ending in a Segmentation Fault, with the code 201041.

V. Conclusions

References

[1] D. Holler, G. Behnke, P. Bercher, S. Biundo, H. Fiorio, D. Pellier, e R. Alford, “HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems,” 2020. [Online]. Disponível: https://bercher.net/publications/2020/Hoeller2020HDDL.pdf. [Acessado: 12-02-2023].

[2] R. Wilson, “Arts & Culture Sudoku number game,” 2023. [Online]. Disponível: https://www.britannica.com/topic/sudoku. [Acessado: 12-02-2023].