On Using Monte-Carlo Tree Search to Solve Puzzles

Mohammad Sina Kiarostami
Center for Ubiquitous Computing, Faculty of ITEE, University of Oulu
Oulu, Finland
mohammad.kiarostami@oulu.fi

Mohammadreza Daneshvaramoli
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
daneshvaramoli@ipm.ir

Saleh Khalaj Monfared
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
monfared@ipm.ir

Aku Visuri
Center for Ubiquitous Computing, Faculty of ITEE, University of Oulu
Oulu, Finland
aku.visuri@oulu.fi

Helia Karisani
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
h.karisani@ipm.ir

Simo Hosio
Center for Ubiquitous Computing, Faculty of ITEE, University of Oulu
Oulu, Finland
simo.hosio@oulu.fi

Hamed Khashehchi
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
hkhashehchi@ipm.ir

Ehsan Futuhi
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
efutuhi@ipm.ir

Dara Rahmati
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
dara.rahmati@ipm.ir

Saeid Gorgin
School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM)
Tehran, Iran
gorgin@ipm.ir

ABSTRACT

Solving puzzles has become increasingly important in artificial intelligence research since the solutions could be directly applied to real-world or general problems such as pathfinding, path planning, and exploration problems. Selecting the best approach to solve puzzles has always been an essential issue. Monte-Carlo Tree Search (MCTS) has surged into popularity as a promising approach due to its low run-time and memory complexity. Thus, it is required to know how to employ this method to solve the puzzles.

In this work, we study the applicability of MCTS in solving puzzles or solving a puzzle with MCTS, not comparing many MCTS approaches. We propose a general classification of puzzles based on their features. This classification consists of four primary classes that provide a mathematical formula for each and their satisfactory criteria. This classification let us know how to utilize MCTS based on the puzzle's features. We pass each puzzle to an MCTS algorithm as a series of satisfaction functions based on this mathematical formulation. The classification can perform general pathfinding or path-planning if the outlining problem is defined within the described mathematical constraints. MCTS progressively solves a puzzle until the functions are completely satisfied in our proposed classification. We examine different puzzles for each class using our proposed methodology. Furthermore, to evaluate the proposed method's performance, each of these puzzles is compared with their available SAT solvers using the Z3 implementation and different variations of MCTS that are generally used.

ICCTA 2021, July 13–15, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9052-1/21/07…$15.00
https://doi.org/10.1145/3477911.3477915

CCS CONCEPTS

• **Theory of computation → Solution concepts in game theory**; *Representations of games and their complexity*; *Theory of randomized search heuristics.*

KEYWORDS

Puzzle, Games, Classification, Monte-Carlo Tree Search (MCTS), SAT.

ACM Reference Format:
Mohammad Sina Kiarostami, Mohammadreza Daneshvaramoli, Saleh Khalaj Monfared, Aku Visuri, Helia Karisani, Simo Hosio, Hamed Khashehchi, Ehsan Futuhi, Dara Rahmati, and Saeid Gorgin. 2021. On Using Monte-Carlo Tree Search to Solve Puzzles. In *2021 7th International Conference on Computer Technology Applications (ICCTA 2021), July 13–15, 2021, Vienna, Austria.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3477911.3477915

1 INTRODUCTION AND BACKGROUND

Throughout the last five decades, proposing a generic, unified, and efficient solver capable of solving any puzzle has been an open problem in Artificial Intelligence research [27]. In this context, pioneered by works in [25] and [30], *General Game Playing* (GGP) strives to introduce an accurate and efficient game-independent platform for Game or Puzzle descriptors. Various works regarding the general expression of puzzles have been proposed. *Game Description Language* (GDL) [22], as a known framework, provides a representation capable of describing multi-player deterministic games. *Regular Board Game* (RBG) [20], a new GGP reasoner and language class, is based on mathematical formalization and proffers efficient reasoning for a wide range of deterministic multi-player board games. However, due to the wide variety of puzzles of different nature, it has been a considerable challenge to suggest a general framework to include all kinds of games rather than board games. Furthermore, from computation complexity and puzzles' features, constructing a unified algorithm is a challenging problem.

Many efforts have been made to investigate the computational complexity of different puzzles [17]. Others have tried to give a comprehensive classification for puzzles [7]. Among all of these classes, most *logic puzzles* are proved to be NP-complete [11], at the same time, completely different *sequential puzzles* such as *Samegame* (*Clickomania*) are also shown to be NP-complete [5]. Nevertheless, for solving puzzles, the issue remains crucial, mainly due to the lack of a single general algorithm to tackle them all. Different puzzles require different heuristics to be effectively solved.

From the perspective of generalized solving algorithms, Monte-Carlo Tree Search (MCTS), as thoroughly surveyed by [8], as a powerful randomized algorithm, is known to be very effective against puzzles in general domains [32],[15]. MCTS has four stages. It employs Monte-Carlo-based simulations to obtain the best final goal in an iterative mechanism. At the *Selection* stage, a constructed tree is traversed among the possible *Children* to find the best *Child* based on the existing *Tree Policy*. It then *Expands* the chosen node for every possible child, guaranteeing a visitation for all non-terminal children of the selected node. Furthermore, multiple randomized *Simulation* are performed, down into the last leaf of the tree, resulting in a *Value* which is calculated via a pre-defined *Value Function*. Finally, as the forth stage, the obtained *Value* affects all the existing nodes a *Back Propagation* procedure.

1.1 Our Contributions

. In this work, we investigate the applicability of MCTS on single-player logical puzzles. We classify the puzzles based on their characteristics into four classes, where MCTS could be applied efficiently or not for each specific *Class*. By defining a description, regardless of the puzzle's domain and definition, mathematical constraints based on the proposed classification could be passed into the unified MCTS. We introduce the classes with their mathematical definition in section III. Also we investigate and apply several optimizations and improvements to MCTS, namely *Ordering*, *Reliable MCTS*, *Fast Rollout*, and *Accurate Randomness* which are explained in section III as well. Ultimately, we evaluate our classification in section IV on several sample puzzles with numerous experimental test cases. We show that the MCTS could be more powerful and efficient in most benchmarks and puzzles than state-of-the-art SAT/CSP solvers.

2 EXISTING WORKS

Regarding the *Logic Puzzles*, it is shown that state-of-the-art SAT solvers are suitable choices for tackling these puzzles. [10]. Similarly, *Constraint Satisfaction Programming* (CSP) uses an identical method to approach these puzzles [26]. Moreover, works employing stochastic CSPs (SCSP) such as [19] are demonstrated to be effective in multi-player games where randomness and stochastic information involved. Recently, authors in [28] present a puzzle solver based on *Luddi* [29], a newly proposed GDL which uses Monte-Carlo Tree search (MCTS) as its core for AI move planning. Incorporating the XCSP3 [6] solver provides a universal integrated framework for constraint problem representation, numerous *Logic Puzzles* are demonstrated to be effectively solved.

On the other hand, MCTS methods studied in [8] are considered one of the most successful approaches to solving puzzles when no domain-specific knowledge is required. MCTS has already been shown to be suitable for multi-player games. [33]. It is believed that MCTS outperforms any other approaches in multi-player games. Improvements in *Computer Go* by the employment of MCTS have led to a migration from conventional Computer Chess to Computer Go [21]. Other approaches mostly suffer from the large size of the problem and become ineffective. MCTS has also been studied for single-player games/puzzles such as *Samegame* [32] and *Sudoku* [9]. However, MCTS solutions for single-player puzzles are not considered favourable and effective as they are in sophisticated multi-player games like *Go* and *Chess*. This fact has motivated the authors to comprehensively investigate the applicability of MCTS to single-player puzzles, where, despite the probabilistic nature of MCTS, the problem should be solved deterministically and accurately. It is also worth mentioning that there are numerous previously proposed heuristic algorithms specifically for each *logic puzzle*. Existing works on GGP solving with MCTS mainly focus on multi-player games where randomness and stochastic procedure are involved. Although MCTS is utilized for *logical puzzles*, by the appearance of high-performance SAT/CSP solvers, logical deterministic puzzles are deemed to be efficiently solved by constraints solvers.

3 IMPLEMENTING CONSTRAINT-BASED MCTS FOR PUZZLE SOLVING

This section defines a general classification for MCTS to understand how we should apply this method to solve a logic puzzle based on the puzzle's features.

3.1 Puzzle Classification

It is already demonstrated in [12], that numerous puzzles could be described by CSP (Constraint Satisfaction Problem) and SAT formulation. However, to efficiently employ the MCTS approach, the SAT description should be modified and simplified to its essential parts. A unified mathematical foundation could be defined for each puzzle. Here, we classify the puzzles into *Three* main *Classes* based on their features and solutions. It is also worthy to emphasize that the described classification could be used to quickly implement the SAT/CSP code.

3.1.1 Class A. The puzzles in *Class A* are defined by the 3-tuple $\mathcal{G} = \langle X, D, C \rangle$. The tuple $X = (X_1, X_2, ..., X_N)$ represents the *Variables*. The *Domain* of each variable is defined by the tuple $D = (D_1, D_2, ..., D_N)$. Thus, the range of the *variable* X_i is restricted by D_i. $(\forall i : X_i \in D_i)$. The *Constraints* are defined by set of L constraints of $C = \{C_1, C_2, ..., C_L\}$, where each constraint is defined by $C_i = (f_i, d_i)$. Boolean function $f_i()$ is a primitive function and d_i with the size of m_i, identifies the domain of *Variables* for $f_i()$. The function $f_i()$ is satisfied over the domain S_i, described as follows:

$$\begin{aligned} &\forall i \in \{1, 2, ..., L\} : \\ &d_i = \{\alpha_1, \alpha_2, ..., \alpha_{m_i}\}, S_i = \{X_{\alpha_1}, X_{\alpha_2}, ..., X_{\alpha_{m_i}}\} \\ &\Rightarrow f_i(S_i) = True \end{aligned} \tag{1}$$

Hence, the puzzle is considered to be solved if :

$$\bigwedge_{i=1}^{L} f_i(S_i) = True \tag{2}$$

The tuple X represents the unknown values to be found in the puzzle. (e.g $X = (X_1, X_2, ..., X_{81})$ in the case of *Sudoku*). Alternatively, if possible, characterizing the puzzle map by a $M \times M$ grid, one could identify the X values in a 2-D representation. (e.g. $X_{i,j}$ is located at the i^{th} row and j^{th} column in the puzzle grid). The *Domain* could also be explained as the permitted *Pieces* or *Values* at each specific position. (e.g $D_i = \{1, 2, ..., 9\}$ is the domain of each cell in *Sudoku*). Most of the logic grid puzzles such as *Sudoku* and Kakuro are generally categorized in the *Class A*. In contrast, other puzzles which involve time-step movements could not be simply included in this class. This kind of puzzles will be mathematically described in the next class.

The puzzles categorized in the *Class A* are known to be solved statically with simple *Constraints*. More importantly, this class of puzzles is *Time Independent*. It demonstrated that these puzzles are usually solved by SAT/CSP solvers (e.g., Z3, XCSP). Nevertheless, we demonstrate that a suitable MCTS organization to approach this category of the puzzle would be as efficient and even faster in some scenarios.

3.1.2 Class B. The *Class B* of puzzles is directly affected by the step moves. To define these puzzles, *Time* dimension should be carefully considered to express an accurate formulation. For instance the *flood it* [2] puzzle is categorized in this *class*. This class could be defined by 6-tuple $\mathcal{G} = \langle X^t, D^t, T^t, V^t, G, E^t \rangle$. Similar to the previous class, the X^t represents *Variables* in the *time steps* $t = \{0, 1, 2, .., t_{final}\}$, where $t = 0$ and $t = t_{final}$ denote the initial and final states, respectively. D^t demonstrates the *Domain* of variables at each time steps. As an important indicator, T^t describes the *Transition* tuple for each variable. The tuple $T^t = \{T_1^t, T_2^t, ..., T_N^t\}$ is responsible for the next states (time-states) of the variables based on the previous variables. It is defined as follows:

$$T_i^t = \{h_i^t(X^{t'}) | t' \leq t, h : D_i^{t'} \rightarrow D_i^t\} \tag{3}$$

Note that h could be any simple function such as *Summation*, *All different*, Etc. Moreover, $E^t = \{E_1^t, E_2^t, ..., E_M^t\}$ illustrates the *Event*, and E_i^t is defined by (p_i^t, q_i^t), where p_i^t is the index and q_i^t is the value of a variable which is altered as an *Event* in the time state t. A transition is valid if the following is satisfied:

$$\begin{aligned} &\forall i \in \{1, 2, 3, ..., n\}, \forall t \in \{1, 2, 3, ..., t_{final}\} : \\ &a_i^t = (X_i^t \in D_i^t) \wedge (X_i^t \in T_i^t) \wedge \\ &((X_i^t = X_i^{t-1}) \vee ((i, X_i^t) \in E^t \\ &\wedge \nexists Y, (i, Y) \in E^t))) = True \end{aligned} \tag{4}$$

In Equation (4), a *Transition* is valid, either if the variable has not been altered from the previous state or has been indicated in the *Event* set. Furthermore, the last term of the Equation ensures that the alteration is due to the occurred *Event*. To ensure that the *Event* is validated and does not violate the puzzle's rules, tuple $V^t = \{V_1^t, V_2^t, ..., V_P^t\}$ with the definition of $V_i^t = (f_i^t, d_i^t)$ is applied as constraints at each time step (similar to *set C* in *Class A*).

Understandably, tuple G, defined by L constraints of $G = \{G_1, G_2, ..., G_L\}$ indicates the *Goal* situation. Each goal constraint is described by $G_i = (g_i, u_i)$. Boolean function $g_i()$ and its domain u_i with the size of m_i, represent *Goal* circumstances for each *Variable*. The puzzle is solved if the goal constraints are satisfied with validated *Transition* as demonstrated in the following:

$$\begin{aligned} &\exists\, t_{goal} \in \{0, 1, 2, .., t_{final}\} : \forall t_j \leq t_{goal} \\ &\forall i \in \{1, 2, ..., L\}, \forall k \in \{1, 2, ..., P\} : \\ &u_i = \{\alpha_1, \alpha_2, ..., \alpha_{|u_i|}\}, d_k^{t_j} = \{\beta_1^{t_j}, \beta_2^{t_j}, ..., \beta_{m_k}^{t_j}\} \\ &S_i = \{X_{\alpha_1}^{t_{goal}}, X_{\alpha_2}^{t_{goal}}, ..., X_{\alpha_{|u_i|}}^{t_{goal}}\} \\ &O_k^{t_j} = \{X_{\beta_j}^{t_j}, X_{\beta_2}^{t_j}, ..., X_{\beta_{m_k}}^{t_j}\} \\ &(\bigwedge_{t_j=1}^{t_{goal}} a^{t_j} \wedge \bigwedge_{j=0}^{t_{goal}} (\bigwedge_{k=1}^{P} f_k^{t_j}(O_k^{t_j})) \wedge \bigwedge_{i=1}^{L} g_i(S_i)) = True \end{aligned} \tag{5}$$

As discussed, the puzzles in the B category are much more complex compared to *Class A* due to the involvement of time step. As an example, *15-puzzle* [34] is categorized in the *Class B* since the puzzle is based on time-state sliding movements.

3.1.3 Class AB. This class is not a complete separated class. It means that the puzzles in this class can be defined and solved by *Class A* or *Class B*, as the problem solver prefers. It is also worth mentioning that the formulation could express any *A* puzzle in in *Class B*. However, some puzzles with dynamic nature can be classified in both classes. The definition and constraint may be different in these two classes for these puzzles. For instance, *NumberLink*[18] puzzle could be described differently in both classes. We refer to these puzzles as *AB Class* puzzles.

3.1.4 Class C. Other puzzles which are not classified in *Class A*, *Class B*, and *Class AB* are left to *Class C*. These puzzles are often more complex and involve random input variable at each time-states. As an example, *2048 game* is labelled in this category since the player faces with random inputs, for instance random 2 or 4. More importantly, these puzzles are usually solved by heuristic algorithms, and pure probabilistic algorithms are not quite successful in solving these puzzles [31]. Also, SAT solvers could not approach these puzzles efficiently since the random parameters increase the memory/time complexity exponentially. However, incremental SAT approaches such as [24] could be employed to solve such puzzles.

In this class, puzzles have at least one random feature that fundamentally makes them unsolvable on a sheet of paper. In every state of these puzzles, one or several random factors exist, making the problem description a stochastic problem that could not be solved efficiently via MCTS. To have a better understanding, consider *2048 game* as an NP-Complete [1] sliding block puzzle. In this puzzle numbered cells with values of 2 or 4 are randomly added each step. The objective is to slide the cells on the grid horizontally or vertically to add them up, creating a cell to achieve the value of 2048 (or more). The puzzle continues until there are no more possible moves. It is clear that a randomized algorithms such as MCTS are deemed inefficient when applied to such puzzle [31]. So, we only describe this class to inform other to not consider MCTS as a promising approach to solve these kinds of puzzles.

3.1.5 Summary. We categorize several famous puzzles based on our classification. As indicated in Table 1, each puzzle that is involved in both *Class A* and *Class B*, is also classified as *Class AB*. Table 1 demonstrates a classification of some common logical puzzles based on the proposed *Classes*.

Table 1: Classification of common logical puzzles based on our proposed *Classes*.

Puzzle	Class A	Class B	Class AB	Class C
Sudoku	X			
Kakuro	X			
Tetris				X
Clickomania		X		
Flood-it		X		
NumberLink	X	X	X	
Slitherlink	X	X	X	
Nonogram	X	X	X	
n-Puzzle		X		
Sokoban		X		
2048				X
Light-up	X			

3.2 Generic MCTS for solving puzzles

Here, we construct a modified MCTS to solve the puzzle based on the classification parameters discussed in the previous section. All the stages of the MCTS, namely *Selection*, *Expansion*, *Simulation*, and *Back-Propagation* with the proposed methodology are explained and discussed. We refer readers to [8] for a complete explanation of MCTS. Based on the classification of puzzles, each puzzle is reduced to several constraint functions and domains.

By describing the puzzle in *Class A*, with the tuple of $\langle X, D, C \rangle$, the goal is to find the correct X_is over domain D_is, to satisfy the constraints C_is. For this structure, each *state* of the MCTS is constructed based on one permitted value for X_i according to the D_i. Hence, in the stage of the *Expansion*, $|D_i|$ number of *Child* nodes are constructed for a *Leaf* node for considered variable X_i. (*Branch Factor* = $|D_i|$) This procedure is demonstrated for a simple 4×4 *Sudoku* in Fig 1.

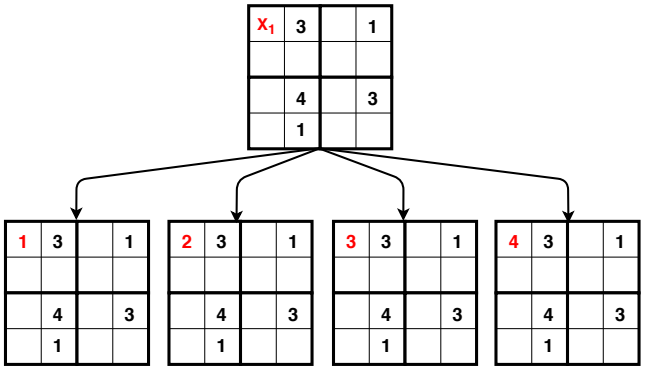

Figure 1: *Expansion* in the proposed MCTS for a simple 4×4 *Sudoku*

Note that the configuration of the children could be arranged differently. However, here we propose the simple *ordered* configuration where all children nodes are a variation of the same X_i. The impact of the different configurations will be discussed as an optimization later on. As for the *Selection*, the most valued node is chosen similar to the conventional MCTS. Consequently, nodes with most constraints satisfied in C is preferred over other nodes and are selected in the successive MCTS iterations. The *Simulation* phase is executed similarly to the regular MCTS procedure, playing random roll-out moves by assigning random values for the remaining X_is until the puzzle is finished. Subsequently, the value of the final node is calculated according to the Equation 6 as follow:

$$Value_{sim} = \frac{\sum_{i=1}^{L}(f_i(S_i))}{L} \tag{6}$$

Afterwards, in the *Backpropagation*, the value of the terminal node, which is computed in the third step, is used to update all previous nodes' value. In other words, the value is propagated toward the top of the tree. In this process, the average value of all children's values is computed and then is propagated to their father. Finally, all of the values, including the value of the root, are updated based on the following:

$$Value = \frac{(K-1) \times Value + Value_{sim}}{K} \tag{7}$$

It is worth noting that the K is the number of times that the tree has visited the node. The stages are executed i times iteratively, similar to the main MCTS algorithm, until the desired X is found as the final solution for the puzzle. With the given mathematical description of *Class B* puzzles, the employment of MCTS is much more sophisticated and complex compared to *Class A*. Due to these

class's time-state nature, the *Branch Factor* of the MCTS is directly affected by the T_i^t, in which the *Transitions* are chosen. t_{final} is passed to the simulation process as the depth of the tree. Moreover, the *Events* are defined in the state relations in the nodes. By incorporating the same methodology explained for *Class A*, the set of *Goals* are validated in the value function when the simulation step is carried out att_{final}. Note that the boolean characteristics of the *Goal* constraints are converted to a dynamic fractional value when the simulation is executed many times.

3.3 Optimizations and Modifications to MCTS

In this section, we have proposed the optimizations and modifications which is employed in the solving procedure.

3.3.1 Order of Solving. In solving process, we have considered an arrangement for choosing the next node for the expansion step [8]. The process which renders the order of children nodes for expansion could be performed before execution of the algorithm as pre-processing, or it could be calculated in the middle of the MCTS procedure. If the simultaneous expansion is employed, the *branching factor* of the tree would be reduced, decreasing the size of the non-simulated section of MCTS [18]. Furthermore, this reduction improves the accuracy of the algorithm. In summary, this optimization reduces the branching factor by choosing a generic approach that lets the algorithm decide to start from where to solve the entire puzzle.

3.3.2 Reliable MCTS. The purpose of this modification is to find the complete solution to the puzzle. In other words, by adding this modification, MCTS would not be halted until it could solve the puzzle. Thus, MCTS would be reliable with 100% accuracy, but most probably, the algorithm will take more time to solve the entire puzzle or even we will not know when it would be stopped. This modification only provides us with the highest accuracy. In *Reliable MCTS*, if any non-terminal node returns a low value, the algorithm will not expand the node. Instead, it prunes the tree, which means it goes back through the tree and continues from there. This modification is similar to employment of *minimax* in MCTS [4].

3.3.3 Fast Rollout Function. Rollout Function is a stage in the simulation step of the MCTS algorithm. This function creates the previous state of the puzzle in simulation, which causes a $O(N^2)$ complexity for each node and then performs the new changes. In *fast rollout function*, a general state is created in the first node of the simulation, and all the following next simulations only apply their changes by $O(1)$ complexity to the original shared state [13].

3.3.4 Accurate Randomness. This optimization is applied to the simulation step of MCTS. Since the simulation step is entirely random based on the definition of MCTS, the algorithm should be repeated for reasonable times to obtain an accurate answer. In this optimization, instead of performing a complete random simulation, we consider some heuristics and validate them after each random move to direct the simulation without adding a considerable computation. This trivial verification impacts the accuracy of the random simulation process significantly. It is usually not possible to find the desired leaf in the tree after 1000 searches. However, considering a minor heuristic and validation (such as choosing the nodes orderly), it might be possible to achieve the correct answer.

4 RESULTS AND EVALUATION

We evaluate the proposed MCTS method on three different versions along with an implementation of SAT solver for each test-case. Our code used for this benchmark is compiled by Java 12 platform and is executed on Ubuntu 16.04 with 8 Intel XEON E5620 CPUs clocked at 2.4 GHz. We execute the programs on a single core with 12GB available RAM. All the results are an average of 50 times executions.

4.1 Target Examples (Puzzles)

Here, we apply our MCTS algorithm to a set of commonly known puzzles. The proposed generic MCTS for described classes is applied to at least one puzzle for each class. The chosen puzzles in our evaluation, span three classifications to experiment results for five puzzles in different categories, and are all equally NP-Complete [17]:

- **Class A:** *Sudoku*, *Calcudoku*, *Light-up*.
- **Class B:** *Clickomania*.
- **Class AB:** *Slitherlink*.

4.2 Results for Class A

In this class, in the MCTS methods, the starting state of the puzzle is placed in the root, and the algorithm chooses the next possible state by filling an empty variable in the puzzle. This is done by expanding the root and selecting a node that represents an empty cell. In this class of puzzles, the solution could be provided by the solver regardless of time, meaning that a step by step procedure to solve the entire puzzle is not required. By the way of example, we investigate *Sudoku* puzzle, *Calcudoku* and *Light-up*, in this class. Here, we apply the proposed MCTS with the discussed modifications and optimizations and compared the accuracy and run-time results to the puzzles' SAT(Z3) solution.

4.2.1 Sudoku. Sudoku is an NP-Complete [17] grid-based number-placement puzzle. One way to solve this puzzle is to solve the mini-grids of size 3×3 squares to significantly decrease the number of total possible permutations and then apply a guessed-free backtracking algorithm[23]. The definition of *Sudoku* puzzle constraints based on the described *Class A* formulation is thoroughly explained in the Appendix of the paper. Each puzzle's constraints follow the similar procedure and can be straightforwardly defined based on the explained classes' mathematical definition. Figure 2 represents run-time and memory consumption of different implementations. As shown, a run-time/accuracy trade-off could be confirmed. The *Standard MCTS* which is the basic implementation of the MCTS method based on the proposed formulation of *Class A*, performs well in run-time compared to corresponding SAT solver solutions. *Optimized MCTS* is the approach where the proposed optimizations are equipped. In this approach, run-time is considerably improved compared to others. The *Reliable MCTS* on the other hand, guarantees a correct final solution for the puzzle by a locked iterative execution of MCTS with a final 100% accuracy constraint. Also, in terms of accuracy, SAT and *MCTS-Reialble* methods solve the

puzzles completely by nature, where the other two might not give a complete final solution.

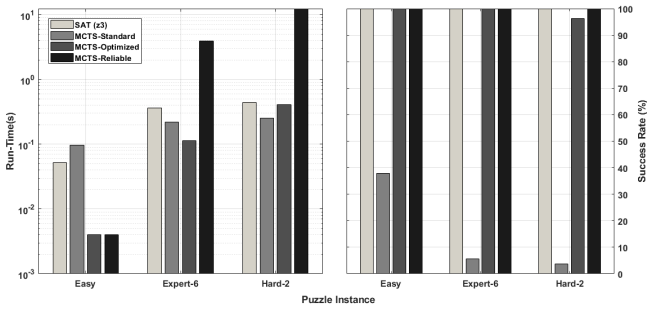

Figure 2: *Sudoku* Evaluation. (Time and Accuracy of four different approaches are compared)

4.2.2 Calcudoku. *Calcudoku* or *Ken Ken* is a grid-based puzzle that divides the grid into different sections with random shapes. This puzzle is a more complex variation of *Kakuro* which is known to be NP-Complete [17]. Figure 3 illustrates the evaluation of solutions for 3 different configuration of the *Calcudoku* puzzle. As indicated, our SAT solver implementation fails to give a solution for the Difficult-8×8 test-bench surpassing the time-limitation of 300 seconds. Due to the time-consuming iterative approach in the *Reliable MCTS*, the overall run-time is higher than *Optimized MCTS* and *Standard MCTS* in the D-8 benchmark. However, MCTS variations are broadly accurate without a significant difference in performance.

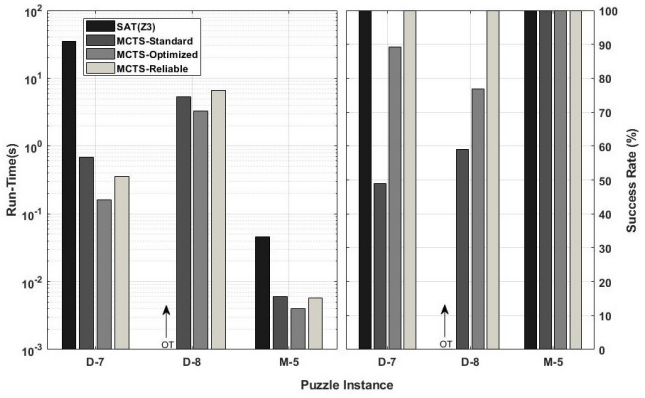

Figure 3: *Calcudoku* Run-time and Accuracy Evaluation.

4.2.3 Light-up. *Light-up* or *Akari* is an NP-Complete [17] grid-based puzzle in which its cells are colored black and white. The goal is to illuminate the map completely while the constraints are satisfied and no light bulb overlapped with each other in emitting light. The most recent work utilizes a *Hopfield* neural network to solve the *Light-up* puzzle [14]. In Figure 4, our implemented solutions are shown. Similar to other puzzles, comparison among different MCTS methods for run-time and accuracy are justifiable. As could be inferred, the SAT solution performs relatively well in

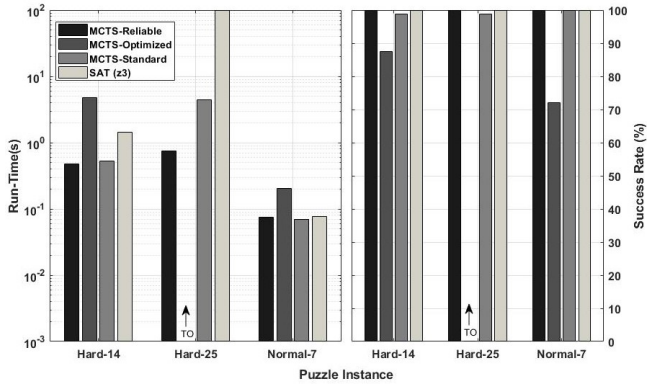

Figure 4: *Light-up* Run-time and Accuracy Evaluation.

normal and small-sized hard puzzles since the logical SAT formulation of this puzzle requires a relatively small number of variables. However, when the puzzle's size is increased to 25, the solution is over-timed since the satisfiability mechanism increases exponentially in memory and computation.

4.3 Results for Class B

In *Class B*, the MCTS algorithm begins from the initial state of the puzzle where no plays have been made. MCTS assigns the next move for the player whenever it can move forward through the tree. The most important aspect of this class's process is that the solution is executed in a step-by-step process analogous to a term as time. It means that the movements perform in turn or round, and each step depends entirely on the previous one. Thus, in these puzzles, solvers have to demonstrate a gradual solution. We have studied the *Clickomania* as a suitable nominee for *Class B* puzzles. *Clickomania* is an NP-Complete [17] puzzle in a grid map with the cells filled with various colours. At each step of the game, the goal is to find and match monochromatic, adjacent cells, which lead to removing the selected cells and falling tiles. The MCTS approach is used recursively to find the best rollout policy for higher-level search in solving this puzzle [3].

Here, we apply *Class B*'s description on the proposed generic MCTS and investigate different optimization, as shown in Figure 5. As discussed earlier, for the *Class B* puzzles, SAT solutions fail as an efficient approach mainly due to the wide search space caused by the *time* parameter. Hence, only the MCTS evaluation are shown. Again, it is evident that the *Optimized MCTS* performs very well in run-time compared to the other two approaches as depicted in Figure 5.

4.4 Results of Class AB

Puzzles in this class could be solved with the approach of *Class A* or *Class B*. In other words, the solvers of these puzzles could perform in both single tree search and step-by-step process. As demonstrated, pathfinding puzzles are categorized in this context. For example, in the *Numberlink*, the player could start solving with each pair of numbers. Also, connecting each pair is not a one-step movement for the algorithm. We examine the puzzle *Slitherlink* in this category.

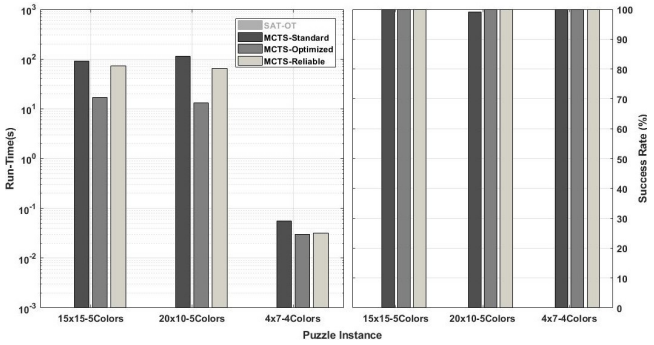

Figure 5: *Clickomania* Run-time and Accuracy Evaluation.

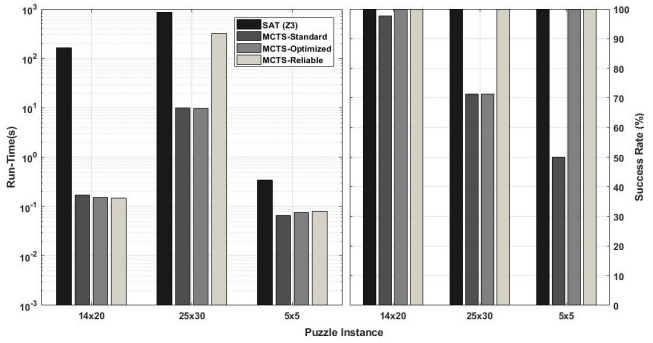

Figure 6: *Slitherlink* Run-time and Accuracy Evaluation.

Slitherlink is an NP-Complete [17] logic puzzle with the final purpose of connecting the adjacent dots, aiming to create a loop without a loosed corner. By employing SAT-solving and Constraint Logic Programming, a new rule-based approach was represented to solve *Slitherlink* in [16]. The comparison of the proposed method and SAT-based solutions in terms of accuracy and run-time are illustrated in Figure 6. As mentioned in the definition of this class, the MCTS approach can be defined and formulated based on *Class A* or *Class B* by choice. In this example, we solve the puzzle based on *Class B* since the number of total variables of the such problem will be much higher in *Class A* representation.

Table 2: Detail evaluation results of solution methods for puzzles based on proposed *Classification*

Solution Method			SAT		*Standard MCTS*		*Optimized MCTS*		*Reliable MCTS*	
Classification	Puzzle	Instances	Accuracy (%)	Run-time (S)	Accuracy (%)	Run-time (S)	Accuracy (%)	Run-time (S)	Accuracy (%)	Run-time (S)
Class A	*Sudoku*	E-1	100	0.0524	37.85	0.097	100	0.004	100	0.004
		N-1	100	0.797	8.14	0.173	100	0.008	100	0.008
		H-2	100	0.443	3.66	0.251	96.22	0.411	100	12.377
		H-3	100	0.810	4.37	0.271	100	0.801	100	0.828
		Ex-4	100	0.341	5.32	0.229	100	0.037	100	0.036
		Ex-5	100	0.336	4.29	0.229	100	0.049	100	0.08
		Ex-6	100	0.359	5.55	0.219	99.11	0.114	100	3.941
	Light-up	N-7	100	0.075	71.98	0.205	100	0.07	100	0.077
		H-7	100	0.071	100	0.284	100	0.074	100	0.075
		N-14	100	0.451	75	3.14	100	0.213	100	0.212
		H-14	100	0.478	87.5	4.739	98.75	0.528	98.75	1.432
		E-25	100	0.714	71.21	79.62	99.87	0.705	99.87	0.683
		N-25	100	0.781	70.42	76.45	99.09	5.327	T/O	T/O
		H-25	100	0.755	T/O	T/O	98.84	4.38	98.84	99.203
	Calcudoku	M-5	100	0.046	100	0.006	100	0.004	100	0.0058
		M-6	100	0.057	64.42	0.12	100	0.026	100	0.0465
		D-6	100	0.139	66.66	0.23	100	0.0499	100	0.1314
		M-7	100	35.21	74.2	0.34	97.3	0.086	100	0.1014
		D-7	T/O	T/O	48.9	0.69	89.1	0.162	100	0.3602
		M-8	T/O	T/O	57.3	1.7	77.3	0.206	100	0.3583
		D-8	T/O	T/O	59.0	5.34	76.9	3.27	100	6.6105
Class B	*Clickomania*	4x7-4C	N/A	N/A	100	0.056	100	0.03	100	0.0321
		13x15-3C	N/A	N/A	100	29.375	100	0.499	100	3.67
		20x10-2C	N/A	N/A	100	11.84	100	0.363	100	1.719
		20x10-5C	N/A	N/A	99.66	90.159	99.89	13.11	100	65.393
		15x15-5C1	N/A	N/A	99.11	115.51	100	16.062	100	71.47
		15x15-5C2	N/A	N/A	100	103.084	100	15.738	100	64.916
		15x15-5C3	N/A	N/A	99.71	115.3	99.82	17.809	100	85.561
		15x15-5C4	N/A	N/A	99.93	89.884	99.86	16.725	100	73.459
Class AB	*Slitherlink*	5x5	100	0.341	49.91	0.0662	100	0.075	100	0.0792
		7x7-1	100	1.276	93.92	0.067	100	0.073	100	0.0689
		7x7-2	100	1.059	87.62	0.091	100	0.092	100	0.087
		10x10-1	100	4.731	74.43	0.24	98.16	0.198	100	0.192
		10x10-2	100	75.427	99.52	0.12	100	0.106	100	0.102
		14x20	100	163.74	97.63	0.171	100	0.154	100	0.15
		20x20-1	100	261.5	99.73	0.203	100	0.181	100	0.207
		20x20-2	100	227.16	93.18	1.679	99.76	1.192	100	0.414
		20x20-3	100	325.05	89.62	2.2	89.62	2.2	100	3.205
		25x30	100	856.21	71.31	9.912	71.34	9.731	100	324.437

By increasing the puzzle size to 25×30, the run-time for an accurate solution increases as shown for SAT and *Reliable MCTS* methods. However, for smaller puzzles, *Optimized MCTS* outperforms other approaches in terms of run-time with an acceptable accuracy rate. Furthermore, Table 2 gives detailed evaluation results of the methods based on proposed *Classification*. Note that the SAT solution fails to solve some instances where the run-time surpasses the maximum limitation of 300 Seconds. Also, the accuracy of the proposed *Reliable MCTS* can be a little lower than 100% when the user forces the algorithm to be finished in a specific time (to prevent over-timing).

5 DISCUSSION AND CONCLUSION

Efficient and generalized AI-based solutions for multi-player puzzles and games have been extensively studied in recent years. In this context, randomized methods such as MCTS have drawn significant attention to the researchers among all methods and algorithms. In this work, we have studied the applicability of the MCTS approach to solve logical single-player puzzles. To introduce a unified MCTS method for solving puzzles, we mathematically categorized puzzles into four classes based on their characteristics. Using a detailed description of each of these classes, we pinpoint that MCTS could be applied effectively. Our evaluation shows that MCTS performs well both in accuracy and run-time in most logical puzzles compared to the conventional state of the art SAT solvers. Furthermore, multiple optimizations for the proposed generic MCTS are studied to improve the solver's performance. To solve a puzzle with the proposed classification, first, the puzzle should be classified based on our proposed classes. Then, it should be formulated based on the class's definition, constraints, and finally should be passed to MCTS core to be solved.

Multiple essential notes should be mentioned according to the employment of the proposed generic MCTS method. Firstly, we have developed an infrastructural mathematical categorization that could be used to describe puzzles and suitably to be approached by MCTS. One could extend this mathematical description to be utilized as an API to fetch the puzzle parameters directly and automate the solution process based on the MCTS method similar to the work done with *Ludii* as explained in [28]. Secondly, the puzzles categorized in class C could also be approached by dynamic MCTS with the similar approaches introduced in [31]. The mathematical formulation of these class could also be studied as further investigation.

ACKNOWLEDGMENTS

This research is connected to the GenZ strategic profiling project at the University of Oulu, supported by the Academy of Finland (project number 318930), and CRITICAL (Academy of Finland Strategic Research, 335729). Part of the work was also carried out with the support of Biocenter Oulu, spearhead project ICON.

REFERENCES

[1] Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. 2015. 2048 is NP-Complete.
[2] David Arthur, Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach. 2010. The complexity of flood filling games. In *Int Conf on Fun with Algorithms*. Springer, 307–318.
[3] H Baier and Mark Winands. 2012. Nested Monte-Carlo Tree Search for online planning in large MDPs. *Frontiers in Artificial Intelligence and Applications*, 109–114. https://doi.org/10.3233/978-1-61499-098-7-109
[4] Hendrik Baier and Mark HM Winands. 2014. MCTS-minimax hybrids. *IEEE Transactions on Computational Intelligence and AI in Games* 7, 2 (2014), 167–179.
[5] Therese C Biedl, Erik D Demaine, Martin L Demaine, Rudolf Fleischer, Lars Jacobsen, and J Ian Munro. 2002. The complexity of Clickomania. *More games of no chance* 42 (2002), 389–404.
[6] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. 2016. XCSP3: An integrated format for benchmarking combinatorial constrained problems. *arXiv preprint arXiv:1611.03398* (2016).
[7] Cameron Browne. 2015. The nature of puzzles. *Game & Puzzle Design* 1, 1 (2015), 23–34.
[8] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.
[9] Tristan Cazenave. 2009. Nested monte-carlo search. In *21st IJCAI*.
[10] Mehmet Celik, Halit Erdogan, Firat Tahaoglu, Tansel Uras, and Esra Erdem. 2009. Comparing ASP and CP on Four Grid Puzzles.. In *RCRA@ AI* IA*.
[11] Diogo M Costa. 2018. Computational Complexity of Games and Puzzles. *arXiv preprint arXiv:1807.04724* (2018).
[12] Broderick Crawford, Carlos Castro, Eric Monfroy, and Nibaldo Rodrıguez. 2009. Solving constraint satisfaction puzzles with constraint programming. In *Congreso de Int Computacional Aplicada, CIC*.
[13] Mohammadreza Daneshvaramoli, Mohammad Sina Kiarostami, Saleh Khalaj Monfared, Helia Karisani, Keivan Dehghannayeri, Dara Rahmati, and Saeid Gorgin. 2020. Decentralized Communication-less Multi-Agent Task Assignment with Cooperative Monte-Carlo Tree Search. In *2020 6th International Conference on Control, Automation and Robotics (ICCAR)*. IEEE, 612–616.
[14] M Fitzsimmons and H Kunze. 2019. Combining Hopfield neural networks, with applications to grid-based mathematics puzzles. *Neural Networks* (2019).
[15] Romaric Gaudel and Michele Sebag. 2010. Feature selection as a one-player game. In *International Conference on Machine Learning*. 359–366.
[16] Stefan Herting. 2004. A rule-based appr to the puzzle of Slitherlink. *Univ. Kent, UK, Tech. Rep* (2004).
[17] Graham Kendall, Andrew Parkes, and Kristian Spoerer. 2008. A survey of NP-complete puzzles. *ICGA Journal* 31, 1 (2008), 13–34.
[18] Mohammad Sina Kiarostami, Mohammadreza Daneshvaramoli, Saleh Khalaj Monfared, Dara Rahmati, and Saeid Gorgin. 2019. Multi-Agent non-Overlapping Pathfinding with Monte-Carlo Tree Search. In *IEEE Conference on Games*.
[19] Frédéric Koriche, Sylvain Lagrue, Eric Piette, and Sébastien Tabary. 2017. Constraint-Based Symmetry Detection in General Game Playing.. In *IJCAI*. 280–287.
[20] Jakub Kowalski, Maksymilian Mika, Jakub Sutowicz, and Marek Szykuła. 2019. Regular boardgames. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1699–1706.
[21] Kirk L Kroeker. 2011. A new benchmark for AI. *Commun. ACM* 54, 8 (2011), 13–15.
[22] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. 2008. General game playing: Game description language specification. (2008).
[23] Arnab Kumar Maji and Rajat Kumar Pal. 2014. Sudoku solver using minigrid based backtracking. In *2014 IEEE International Advance Computing Conference (IACC)*. IEEE, 36–44.
[24] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. 2014. Ultimately incremental SAT. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 206–218.
[25] Allen Newell, John C Shaw, and Herbert A Simon. 1959. Report on a general problem solving program. In *IFIP congress*, Vol. 256. Pittsburgh, PA, 64.
[26] Barry O'Sullivan and John Horan. 2007. Generating and solving logic puzzles through constraint satisfaction. In *PROCEEDINGS OF THE NATIONAL CONF ON AI*, Vol. 22. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 1974.
[27] Cédric Piette, Eric Piette, Matthew Stephenson, Dennis JNJ Soemers, and Cameron Browne. 2019. Ludii and XCSP: Playing and Solving Logic Puzzles. In *2019 IEEE Conference on Games (CoG)*. IEEE, 1–4.
[28] Cédric Piette, Éric Piette, Matthew Stephenson, Dennis JNJ Soemers, and Cameron Browne. 2019. Ludii and XCSP: Playing and Solving Logic Puzzles. *arXiv preprint arXiv:1907.00245* (2019).
[29] Eric Piette, Dennis JNJ Soemers, Matthew Stephenson, Chiara F Sironi, Mark HM Winands, and Cameron Browne. 2019. Ludii-the ludemic general game system. *arXiv preprint arXiv:1905.05013* (2019).
[30] Jacques Pitrat. 1968. Realization of a GGP program.. In *IFIP congress (2)*. 1570–1574.
[31] Philip Rodgers and John Levine. 2014. An investigation into 2048 AI strategies. In *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, 1–2.

[32] Maarten P. D. Schadd, Mark H. M. Winands, Mandy J. W. Tak, and Jos W. H. M. Uiterwijk. 2012. Single-player Monte-Carlo tree search for SameGame. *Knowl.-Based Syst.* 34 (2012), 3–11.
[33] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529, 7587 (2016), 484.
[34] Jerry Slocum and Dic Sonneveld. 2006. The 15 puzzle book. *Slocum Puzzle Foundation* (2006).

A APPENDICES

A.1 Mathematical Representation of Sudoku Puzzle for Class A

In this section, we explain how we employ the proposed classification and apply the puzzle's constraints as an example to indicate how this classification helps us to see the puzzle from MCTS's point of view. Take a conventional 9×9 *Sudoku* puzzle as the example. Here we described a two-dimensional *variable* for each cell in the puzzle plane as in tuple X. The *domain* for each of these 81 variables are also defined in the tuple D. So, we have:

$$X = (X_{1,1}, X_{1,2}, ..., X_{9,9}) \tag{1}$$

$$D = (D_{1,1}, D_{1,2}, ..., D_{9,9}) \tag{2}$$

According to the definition in a 9×9 Sudoku puzzle, the domain of the variables could be defined as:

$$\forall (i,j) \in N^+_{9,9} : D_{i,j} = \{1, 2, .., 9\} \tag{3}$$

The *Condition* tuple is consist of a tuple of 27 conditions over *rows*, *columns* and *mini-squares* (9 + 9 + 9) described as:

$$C = \{(f_1, d_1), (f_2, d_2), ..., (f_{27}, d_{27})\} \tag{4}$$

The desired f function for each of these conditions is to satisfy the uniqueness feature among the desired domain:

$$\forall i \in N^+_{27} : f_i = alldifferent \tag{5}$$

And finally, the desired domain for the condition functions are divided into three separated group where *rows*, *columns* and *mini-squares* are considered.

$$\begin{aligned} &\forall i \in N^+_9 : \\ &d_i = \bigcup_{j \in N^+_9} \{X_{i,\ j}\}, \\ &d_{i+9} = \bigcup_{j \in N^+_9} \{X_{j,\ i}\}, \\ &d_{i+18} = \bigcup_{j \in N^+_9} X_{3\times\lfloor\frac{(i-1)}{3}\rfloor+\lfloor\frac{(j-1)}{3}\rfloor+1,\ 3\times((i-1)\%3)+(j-1)\%3+1} \end{aligned} \tag{6}$$

Note that the first condition domain is defined for the *rows* variable. The second group is described as the transposed variable domain for the *column* variables. Moreover, the last group defines the mathematical representation for the variables that fall into the *mini-square* domain.

By taking into account the described tuples and variables, the desired assignment of the X tuple could lead to the correct solution for any given *Sudoku* puzzle.