TetrisVs - A Case Study on a Game Playing Agent in Prolog

Iacovazzi Antonio Raffaele
University of Bari Aldo Moro
a.iacovazzi6@studenti.uniba.it

April, 2024

Abstract

The aim of this work is to explore the capabilities of Prolog as a tool for building game-playing agents. In order to do this, a new version of Tetris called TetrisVs was developed. In this version, in addition to the classic Tetris gameplay, a versus mode for two players was added. This new mode is significantly different from the classic versus mode, like the one proposed by Tengen [1], where the two opponents play on two different game boards; in this new version, the game board is shared, and the goal is to clear more rows than the opponents without reaching game over. This allows approaching the game with more complex strategy, giving also the possibility to deal with unseen dynamics when developing the opponent's agent. The proposed agent can handle both the classic Tetris gameplay and the versus mode, essentially using the same algorithms. Furthermore, the proposed algorithms were developed with generality and explainability as a key factor, in order to make it easier to reuse them in other ludic contexts and to make possible to build AI that offers to the player a clear reason behind this its decisions. As a result, the proposed solution serves as a valid foundation for game-playing agents in Prolog, despite some performance-related criticalities.

1 Introduction

Tetris is probably the most famous puzzle video-game in the world. Created by Aleksej Leonidovič Pažitnov in 1986[1], this game never ceased to entertain and challenge players. Even today, Tetris is still subject of interest, both in academia for it's theoretical properties[3] and in the gaming competitive scene, where several player are pushing beyond the limits some classics versions of this game [4] [5]. Given its simple but non-trivial nature, this game has been subject

[1] A humorous, though not entirely historically accurate, reconstruction of the story behind the western distribution of this game can be found in the movie Tetris, directed by Jon S. Baird[2].

of study in the AI field since the subsequent years after its release using several techniques[6]. TetrisVs, the proposed version in this work, adds a new versus mode that open the game to new interesting frontiers both for human players and for AI researchers. In this section will be discussed both the specific rules set of this version and the available implemented game modes.

1.1 TetrisVs Rules an Game Modes

1.1.1 TetrisVs Rotation System

During the last decades, hundreds of different versions of Tetris have been developed[2], each with its own specificity and additional rules. TetrisVs is inspired by the Game Boy version[8] of the game, where there weren't any additional mechanics like wall kick or lock delay. The only difference is that the L, J, and T tetrominoes[3] appear on the screen with their "spikes" pointing upwards. The actual configuration is shown in the following figure.

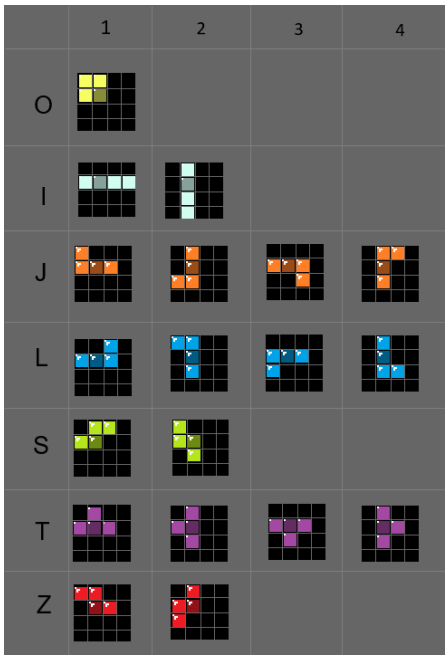

Figure 1: The rotation system used by TetrisVs along with the alphabet letter that identifies it and the number that identifies its rotation state. The fulcrum of each tetromino is also highlighted using a darker color.

[2]In 2017 Guinness World Records reported Tetris to be the game with the most official versions released, more than 220 versions, over at least 70 different platforms[7].

[3]In the Tetris lingo, tetrominoes are identified using the capital alphabet letter they resemble.

All tetrominoes appear within a 20x10 game board grid, with their fulcrum located at coordinates (5,1).

1.1.2 TetrisVs Score System

The score is computed multiplying the number of cleared rows by 100, then a bonus is added using the following rules:

- Bonus for one cleared row: 0.
- Bonus for two cleared row: 100.
- Bonus for three cleared row: 200.
- Bonus for four cleared row: 300.

These bonuses exists in order to give more weight to moves that clear more than one row.

1.1.3 TetrisVs Settings and Game Modes

TetrisVS presents five different game mode:

- Solo: the classic single player gameplay, the game goes on until game over.
- Solo AI: where the Solo mode is played by the Prolog agent.
- Vs Player: where two human players compete on the same board in order to gain more point while avoiding game over.
- Vs AI: where an human player compete in the Vs mode against the Prolog agent.
- AI Vs AI: where two Prolog agents compete in the Vs mode.

Furthermore, players have the option to adjust the following settings:

- AI explanation (enabled or disabled): when enabled, this feature pauses the game after each AI move, allowing players to review the logic behind the move using the browser console log.
- AI opponent level (easy or hard): allows the player to adjust the strength of the AI opponent between two possible levels.
- Vs Mode number of turns (any number between 1 and 5): allows the player to select the number of turns played in Vs mode.

In both Solo and Vs modes, every 60 seconds the level increases and the falling speed of the tetrominoes is augmented by 20%. Each new tetromino is generated randomly and the game enables players to preview the next available tetromino, facilitating the planning of intricate maneuvers. This functionality

holds particular significance in the Vs mode, empowering players to anticipate opponent countermoves and strategically plan their actions.

Figure 2: A portion of the game title screen showing all the settings and game modes.

1.2 Software Architecture of the Game

The software architecture of TetrisVs is based on two main components: the JavaScript client and the SWI-Prolog server. The JavaScript client operates as a stand-alone game when no AI interaction is required, as in the Solo and Vs Player modes. In the other game modes AI interaction is necessary, and then the two components exchange information bidirectionally, as shown in the following figure.

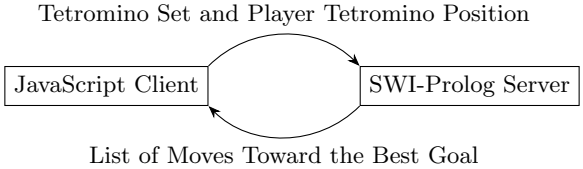

Figure 3: Architecture of TetrisVs

In the Solo AI and AI vs AI modes, the client sends information about the current tetromino set to the server. The server then computes the optimal execution plan and returns a list of moves for the client to execute. In the Vs AI mode, the server also sends information about the position of the player's tetromino on the game board, ensuring that the internal representation of the game board on the server side remains updated. Technical detail about the SWI-Prolog server will be discussed later.

1.3 TetrisVs: A Formal Classification

Before proceeding into the technical details of the implemented game-playing agent, it's essential to categorize the game using some formal definitions. This step is necessary in order to reason effectively about how to develop an agent capable of playing TetrisVs in both of its game modes.

In Solo mode, TetrisVs is classified as a *stochastic* game with *perfect information*. It's stochastic because, at each turn, the next tetromino is generated randomly. It's considered to have perfect information because players can always make optimal moves based on the current state of the game board and the upcoming tetromino.

In Vs mode, the game inherits all the properties of the single-player mode but also qualifies as a *zero-sum game* with cooperative elements. This is because when one player wins, the other necessarily loses. However, both players must also collaborate to find a balance point to avoid a game over, where both players lose simultaneously.

2 Prolog Agent Implementation

The field of Artificial Intelligence is into an era of renewed interest thanks to the outstanding results obtained by the use of machine learning techniques based on Artificial Neural Networks(ANN). In the field of gaming, ANNs are becoming increasingly relevant thanks to deep reinforcement learning[9], where the ANN learns how to win going through a massive number of simulation and adjusting it's own internal weight on the basis of the produced outcome. However, these kind of techniques, despite their exceptional results even in really complex game like Go [4], they acts like black boxes; their produced moves are the result of complex mathematical operations and there is no way to understand a clear reason behind certain decisions.

This is the main reason that make, the old fashioned logic-programming based AI, still a relevant option when developing game-playing agents. Using these techniques it's possible to realize agents that can access to a clearly defined knowledge base about moves and explaining the ratio behind their decision using

[4] An intriguing narrative of the game between Lee Sedol and the Google-developed agent AlphaGo can be enjoyed in "AlphaGo - The Movie," directed by Greg Kohs [10]. An interesting exploration of the stories behind the development of AI agents for seven classic games can be found in "Seven Games" by Oliver Roeder [11].

heuristic evaluation and search algorithms. For these reason Prolog was chosen as main tool for developing the agent behind the TetrisVs AI.

In the following subsections, the primary predicates involved will be discussed. These predicates are essential for implementing intelligent decision-making processes, such as determining optimal moves, evaluating game states, and adapting strategies based on dynamic gameplay conditions. However, it's important to note that due to the significant number of predicates in the project, only the main ones will be examined in this discussion. Auxiliary predicates, while important for supporting the overall functionality of the AI, will not be covered in detail.

2.1 Tetrominoes and Game Board Representation

In this subsection the main predicates representing the fundamentals knowledge of TetrisVs are described. The knowledge about game board dimensions and tetraminoes spawning point was defining using simple Prolog facts.

```
gameBoardW(10).
gameBoardH(20).
tetrominoSpawnX(5).
tetrominoSpawnY(1).
```

The knowledge about all the tetraminoes blocks placed on the game board is expressed by means of the following dynamic predicate[5]:

```
startGbL/1
```

The asserted game board list is accessible through the following rule:

```
getStartGbL(GbL) :-
    startGbL(GbL),
    !.

getStartGbL([]).
```

The variable 'GbL' stands for "Game Board List" and contains, during the reasoning process, the actual list of occupied cells on the game board. The second occurrence of 'getStartGbL/1' is a simple facts that unifies to an empty list if 'startGbl' is not asserted yet.

Similarly, the knowledge about the available tetraminoes to be placed is stored in a list contained in the argument of the following dynamic predicate.

```
tetrominoes/1
```

[5]In Prolog a predicate declared as "dynamic" implies that can be asserted and retracted during the course of reasoning process, this allow to modify the knowledge dynamically and use facts like a global memory.

The knowledge about tetrominoes shapes and rotation was defined as a set of both facts and rules, in the following excerpt the T tetromino is used as example.

```
fitPiece(t1,R1,C1,R2,C2,R3,C3,R4,C4,GbList) :-
    C1 = C3,
    R1 = R2,
    R1 = R4,
eq(C2,C1,-1),
    eq(R3,R1,-1),
    eq(C4,C1,+1),
    fitPiece(R1,C1,R2,C2,R3,C3,R4,C4,GbList).

fitPiece(t2,R1,C1,R2,C2,R3,C3,R4,C4,GbList) :-
    C1 = C3,
    R1 = R4,
    C1 = C2,
eq(R2,R1,+1),
    eq(R3,R1,-1),
    eq(C4,C1,+1),
    fitPiece(R1,C1,R2,C2,R3,C3,R4,C4,GbList).

fitPiece(t3,R1,C1,R2,C2,R3,C3,R4,C4,GbList) :-
    R1 = R3,
    R1 = R4,
    C1 = C2,
eq(R2,R1,+1),
    eq(C3,C1,-1),
    eq(C4,C1,+1),
    fitPiece(R1,C1,R2,C2,R3,C3,R4,C4,GbList).

fitPiece(t4,R1,C1,R2,C2,R3,C3,R4,C4,GbList) :-
    R1 = R3,
    C1 = C2,
    C1 = C4,
eq(R2,R1,+1),
    eq(C3,C1,-1),
    eq(R4,R1,-1),
    fitPiece(R1,C1,R2,C2,R3,C3,R4,C4,GbList).
```

The 'fitPiece/10' allows to represents a tetromino in its different rotation states on the gameboard. It serves both a representational purpose, defining the constraints between compounding blocks, and an operational purpose, checking if the tetromino can actually be placed on the game board. Each rotation state of a tetromino is uniquely identified by a code (such as t1, t2, t3, and t4 in the example) along with the coordinates of its 4 constituent blocks. The coordinates are computed using the pivot point as a reference; indeed, it is the only

point actually manipulated by action predicates that will be discussed later. All other points are computed using an auxiliary predicate called 'eq/3', which performs simple mathematical computations and stores the results in the first variable. Furthermore, after computing all the coordinates of all the blocks, a check is performed to understand if the tetromino can actually be placed on the game board, represented through the variable 'GbList', using the predicate 'fitPiece/9'.

```
fitPiece(R1,C1,R2,C2,R3,C3,R4,C4,GbList) :-
    freeCell(R1,C1,GbList),
    freeCell(R2,C2,GbList),
    freeCell(R3,C3,GbList),
    freeCell(R4,C4,GbList).
```

This predicate is true if each cell, represented as a couple of coordinate, is free to respect of the game board in the variable 'GbList' and coherent with the game board boundaries.

In addition to the information about how to representing a tetromino on the game board, it's also necessary to incorporate the logic for rotation and determine the initial orientation in which a tetromino appears.

```
firstShape(t, t1).
rotation(t, t1, t2).
rotation(t, t2, t3).
rotation(t, t3, t4).
rotation(t, t4, t1).
```

The predicate 'firstShape/2' allows to associate the generic symbol representing a tetromino to its first rotation shape. The predicate 'rotation/3' allows to associate a specific rotation state to it successors. The information about the generic symbol is also included in order to easily transform a rotation shape symbol into its generic version. All the presented predicates was properly replicated to represent all the possible tetrominoes and can be found in the file "tetris.pl" in the source code.

To conclude this subsection, the predicates that find possible available position for a given tetromino are described.

```
findPossibleGoals(T,List,GbList) :-
    findall((T1), rotation(T, T1, _), Lr),
    findPossibleGoals([],Lr,List,GbList).

findPossibleGoals(Found,[],Found,_).

findPossibleGoals(Found,[Ht|T],L,GbList) :-
    findall((Ht,R,C), tetrominoGoal(Ht,R,C,GbList), List),
    append(Found,List,NewList),
    findPossibleGoals(NewList,T,L,GbList).
```

```
tetrominoGoal(Tr,R,C,GbList) :-
    freeCell1(R,C,GbList),
    fitPiece(Tr,R,C,_,_,_,_,_,_,GbList),
    R1 is R + 1,
    \+fitPiece(Tr,R1,C,_,_,_,_,_,_,GbList).
```

The predicate 'tetrominoGoal/4' generates all possible positions for a given tetromino in a certain rotation shape, unifying the coordinates of its pivot point in the variables 'R' and 'C'. To achieve this, the predicate 'freeCell1/3' is called, which, given a game board list, generates all free cells on the game board and unifies their coordinates in the variables 'R' and 'C'. Then, a check is performed to determine if the entire tetromino can fit in that position, followed by a check to determine if the tetromino cannot be placed in the successive row. This sequence of checks corresponds to the following definition: a tetromino goal is defined as coordinates that allow a tetromino to be drawn without allowing it to go further down on the game board.

The predicate 'tetrominoGoal/4' can return multiple coordinates for a given tetromino state in backtracking. Furthermore, we are interested in collecting all possible goals for every rotation state of a tetromino. To achieve this, the predicate 'findPossibleGoals/3' is defined. It takes a generic tetromino symbol in the variable 'T', collects all possible rotation states using the default Prolog predicate 'findall/3', and stores them in the variable 'Lr'. This variable, along with the game board list, is then passed to 'findPossibleGoals/4', which unifies the list of all possible goals for all rotation states in the variable 'List'.

2.2 Game Board Manipulation Predicates

In this subsection, the predicates responsible for operating on the game board in TetrisVs are described. These predicates play a crucial role in managing various aspects of the game board, including the placement of tetrominoes, updating the game board, and ensuring its integrity throughout gameplay.

```
placePiece(T,R1,C1,LGbpre,LGbpost,NumberOfClearedRows) :-
    fitPiece(T,R1,C1,R2,C2,R3,C3,R4,C4,LGbpre),
    L = [[R1,C1],[R2,C2],[R3,C3],[R4,C4]],
    append(LGbpre,L,LGbpre1),
    computeNewGameBoard(LGbpre1,LGbpost,NumberOfClearedRows).

placePiece(T,R1,C1) :-
    getStartGbL(GbListOld),
    placePiece(T,R1,C1,GbListOld,GbListNew,_),
    retractall(startGbL(_)),
    asserta(startGbL(GbListNew)).

computeNewGameBoard(GbListOld,GbListNew,NumberOfClearedRows) :-
    findall((R),clearedRow(R,GbListOld),ClearedRows),
    removeRows(ClearedRows,GbListOld,GbListMid),
```

```
    length(ClearedRows,NumberOfClearedRows),
    shift(ClearedRows,GbListMid,GbListNew),
    !.
computeNewGameBoard(GbListOld,GbListOld,0).

clearedRow(R, GbList) :-
    countOccCelInRow(R,Occ, GbList),
    gameBoardW(W),
    Occ = W.
```

The predicate 'placePiece/7' first verifies whether the tetromino can be placed on the game board. If so, it creates a new temporary game board list using the Prolog predicate 'append/3'. This new list, represented by the variable 'LGbpre1', is then passed to the predicate 'computeNewGameBoard/3', which performs a series of manipulations to construct a new game board list contained in 'LGbpost'. If no rows are cleared during this operation, 'LGbpost' remains the same as 'LGbpre1'. However, when some rows are cleared, a new list of occupied cells is generated, and the variable 'NumberOfClearedRows' represents the number of rows cleared during this operation. The presence of cleared rows is performed by means of the predicate 'clearedRow/2' that verify, for a given row, if the number of occupied cells is equal to the width of the game board.

The predicate 'placePiece/3' simply calls 'placePiece/7' with the position of the placed tetromino and the currently asserted game board list. After this computation, the old game board list is retracted and the new one is asserted. This last predicate is used by the client side to communicate the move of the human player.

2.3 Game Board Evaluation Predicates

A key component in every game playing agent is the evaluation function, which assesses the desirability of the current game state.

In TetrisVs, the evaluation function operates on potential game boards resulting from the application of available moves. The agent evaluates these potential game boards to identify the most promising outcome, guiding the selection of the move that leads to that outcome. This approach enables the agent to make informed decisions and optimize its gameplay strategy in TetrisVs. In TetrisVs the following formula is used to compute the score associated to the evaluated game board:

$$Score = -10 \sum_{i=1}^{GBHeight} (Entropy(Row_i)) + AggrHeight - TotHoles - Bumpiness + RowsCleared$$

And is computed by the following predicate.

```
gameBoardScore(GbList,AggregateHeight,RowsCleared,Holes,Bumpiness,SumEnt,Score) :-
    occCellForColumn(GbList,O4C),
```

```
heightForColumn(GbList,H4C),
sumlist(H4C,AggregateHeight),
totalHoles(H4C, O4C, Holes),
bumpiness(H4C, H4C, Bumpiness),
E is -10,
sumEntropyOfGameBoard(SumEnt,GbList),
Score is E*SumEnt - AggregateHeight - Holes - Bumpiness + RowsCleared.
```

The predicate takes as input the game board list in 'GbL' and returns as output the score along with its components. These values are returned because they are useful for ensuring the explainability of the agent. Let's analyze each component of the formula with reference to the predicate called in the right-hand side of 'gameBoardScore/7' that computes it.

Entropy represents the entropy of a row, computed using its information theory definition [12]. In the formula a summation of the entropy of each row is performed. This approach guides the agent toward moves that lead to completed lines because the entropy can be decreased by either completing an already started line or avoiding the initiation of a new line. This computation is performed by the predicate 'sumEntropyOfGameBoard/2' that computes the entropy row by row and then sums the results. The sum of entropy is multiplied by -10 in the formula to amplify its relevance in the overall computation.

AggrHeight represents the sum of the heights of each column in the gameboard, where the height of a column is defined as the y-coordinate of the highest occupied cell in that column. The objective is to keep this number low as the game progresses. This computation is performed by the predicate 'heightForColumn/2', which returns a list of heights for each column in the gameboard. These heights are then summed using the standard Prolog predicate 'sumlist/2' to obtain the aggregate height.

TotHoles represents the number of holes, defined as any free cell with an occupied cell above it. While this definition may lead to a weaker concept of holes[6] with some practical implications in computing the score (especially in the Vs mode), it was preferred since was simpler to compute leading to a less time consuming elaboration. Despite its simplified definition, minimizing this component is crucial. It is one of the most intuitive elements that also human players try to minimize while playing Tetris.

The computation of this element of the score is handled by the predicate 'totalHoles/3'. This predicate subtracts, for each column, its height by the number of occupied cells, obtaining the number of holes for each of them. These values are then summed up to obtain the global value representing *TotHoles*.

Bumpiness represents the sum of the difference, in absolute value, between the subsequent pairs of columns. A high bumpiness value indicates that the game board contains many uneven surfaces or "wells," which can quickly lead to

[6] In this work, a "weak" hole refers to a free cell positioned directly beneath an occupied one. A "proper" hole denotes a free cell that cannot be reached starting from the tetromino spawn point of the game board. The distinction between the two lies in the accessibility of the hole. In TetrisVs a cell is recognized as an hole when satisfy the "weak" definition.

a loss, so it's important to minimize this component as well. This computation is performed by the predicate 'bumpiness/3'.

RowsCleared is probably the most intuitive component; each move that is actually capable of clearing some rows is a good move, so the number of cleared rows must influence the overall score related to a move. This is the only component that is desirable to maximize in the formula. This component is not computed directly inside 'gameBoardScore/7' because, after the computation of a new game board by means of 'computeNewGameBoard/3' this information is not retrievable anymore. For this reason the value unified in 'NumberOfClearedRows' is then used as input in the variable 'RowsCleared' in 'gameBoardScore/7'.

The overall formula output adheres to the principle of "the greater, the better." In this context, smaller numbers are indicative of poor game board configurations, while numbers closer to or greater than 0 are associated with favorable game board configurations. This principle guides the evaluation of game board states, where the goal is to maximize the overall score by minimizing negative factors and maximizing positive factors.

This formula draws inspiration from the work of Yiyuan Lee [13], but takes a different approach by using the sum of the entropy of each row as the key factor to guide the agent toward optimal moves instead of searching for the optimal set of weights for each component of the score using genetic algorithm, as done in Lee's work. The decision to use the sum of rows' entropy was made for the sake of simplicity, avoiding the need for complex parameter optimization while still achieving a useful and effective heuristic.

2.4 Depth-First Heuristic Planner

The planner presented in this subsection allows the generation of a list of actions that efficiently moves the current tetromino towards the selected goal. As the primary objective of this work is to study predicates that aid in developing AI agents within game-playing contexts, the planner was designed to be easily adaptable for use it in other contexts as well. To achieve this goal, the planner was developed as a standalone Prolog file, making it easily reusable as an importable module. The planner is implemented as a depth-first search algorithm [14] that also utilizes a heuristic function to order the potential child nodes when expanding a non-goal node. This approach was chosen for its simplicity, as it aligns well with the requirements of TetrisVs and is straightforward to implement in Prolog. The implementation is the following:

```
:- module(planner, [planner/7]).
planner(Start, Goal, Actions, Heuristic, Plan, PlanStory, GoalChecker) :-
    planner(Start, Goal, Actions, Heuristic, [Start], [], Plan, [],
    PlanStory, GoalChecker).

planner(CurrentNode, Goal, _, _, _,Plan, Plan, TempPlanStory, PlanStory, GoalChecker) :-
```

```
    call(GoalChecker,CurrentNode,Goal),
    append(TempPlanStory,[Plan],PlanStory),
    !.

planner(CurrentNode, Goal, Actions, Heuristic, VisitedNodes, TempPlan, Plan,
TempPlanStory, PlanStory, GoalChecker) :-
    evaluateActions(Actions,Heuristic,CurrentNode,Goal,OrderedEvaluatedActions),
    append(TempPlanStory,[TempPlan],TempPlanStory1),
    member([_,Action], OrderedEvaluatedActions)
    Z =.. [Action,CurrentNode,SuccessorNode],
    call(Z),
    \+member(SuccessorNode,VisitedNodes),
    append(VisitedNodes,[SuccessorNode],VisitedNodes1),
    append(TempPlan,[Action],TempPlan1),
    planner(SuccessorNode, Goal, Actions, Heuristic, VisitedNodes1, TempPlan1,
    Plan, TempPlanStory1, PlanStory, GoalChecker).

evaluateActions(Actions, Heuristic, Input, Goal, OrderedActions) :-
evaluateActions(Actions, Heuristic, Input, Goal, [], OrderedActions).

evaluateActions([], _, _, _, [], []).

evaluateActions([], _, _, _, TempActions, OrderedActions) :-
sort(1, @>=, TempActions, OrderedActions).

evaluateActions([Action|T], Heuristic, Input, Goal, TempActions, OrderedActions) :-
    Eval = [Action, Input, Goal],
    ExecuteEvaluation =.. [Heuristic,Eval,Score],
    call(ExecuteEvaluation),
append(TempActions, [[Score,Action]], TempActions1),
evaluateActions(T, Heuristic, Input, Goal, TempActions1, OrderedActions).
```

The predicate 'planner/7' is the one exposed to the outside. It takes the following inputs:

- 'Start': a starting node.
- 'Goal': the goal node.
- 'Actions': a set of executable predicates that allow to transform a node into another node.
- 'Heuristic': a predicate that implements a function that measure how much each action leads the current node towards the goal. This predicate must be of arity two with a three element list having the structure '[Action, Input, Goal]' as first argument. This is necessary to be compliant with the 'evaluateActions/6' predicate requirements.

- 'GoalChecker': a predicate that compare the current node with the goal in order to determine if it has been reached.

And give as outputs:

- 'Plan': a list containing the actions that allow to reach the goal.
- 'PlanStory': a list that reports the complete story of the search process.

The 'Start' and 'Goal' nodes can be any Prolog term, such as a simple atom, a list, or a compound term. They may have different structures, as long as the 'GoalChecker' predicate is capable of comparing them effectively[7].

The predicate 'planner/7' calls 'planner/10' which adds 'VisitedNodes', that contains the list of already generated nodes in order to avoid loops, and two temporary lists, 'TempPlan' and 'TempPlanStory', that unifies to 'Plan' and 'PlanStory' when the goal checker predicate recognize that the goal has been reached.

The predicate 'planner/10' actually build the plan executing the following steps:

1. First, it checks if the goal has been reached. If so, it produces the ultimate plan and terminates further computation using a cut (!).
2. If the goal has not been reached, the predicate evaluates the score related to each available action with respect to the current node and the goal using the 'evaluateActions/5' predicate. This produces an ascending ordered list containing '[Score, Action]' elements.
3. The most promising action is selected using the standard Prolog predicate 'member/2', and the action is executed on the current node, producing its successor, 'SuccessorNode'.
4. A check is performed to determine if the new node has already been generated. If not, the new node is added to the 'VisitedNodes' list, producing the new list 'VisitedNodes1', and the selected action is added to the list 'TempPlan', producing 'TempPlan1'.
5. After these steps, 'planner/10' is called recursively and going back to step 1, passing the newly produced lists.
6. If the execution of the action fails or the generated new node is already in the list, the predicate fails and backtracks to the 'member/2' predicate, which considers the subsequent action.
7. If no action can be applied, the entire predicate fails.

The planner module is used by TetrisVS providing the following predicates.

[7] This property will be exploited in the section Appendix A: Snake.

Action predicates:

```
action(rotate).
action(right).
action(left).
action(down).

%listing the actual transformation involved by the moves
rotate((T1,R1,C1,GbL),(T2,R1,C1,GbL)) :-
    rotation(_,T2,T1),
    fitPiece(T2,R1,C1,_,_,_,_,_,_,GbL).

left((T1,R1,C1,GbL),(T1,R1,C2,GbL)) :-
    C2 is C1+1,
    fitPiece(T1,R1,C2,_,_,_,_,_,_,GbL).

right((T1,R1,C1,GbL),(T1,R1,C2,GbL)) :-
    C2 is C1-1,
    fitPiece(T1,R1,C2,_,_,_,_,_,_,GbL).

down((T1,R1,C1,GbL),(T1,R2,C1,GbL)) :-
    R2 is R1 - 1,
    fitPiece(T1,R2,C1,_,_,_,_,_,_,GbL).
```

These action predicates define the movements of tetrominoes in TetrisVS. Notably, the naming convention might seem counter-intuitive, the actual action performed is the opposite of what the predicates' name declares. This design choice is intentional and allows to deal with efficiency complex scenarios, such as T-Spin[15] rotations, where backward search proves to be more effective than forward search. The action predicates are collected though the fact 'action/1' that allows to build easily a list of the action predicates' name through 'findall/3'.

Heuristic predicates:

```
evaluateMovement([rotate, (T1,_,_,_), (T2,_,_,_)], Score) :-
    diff(T1,T2,Diff),
priority(rotate,Priority),
Score is  Diff * Priority.

evaluateMovement([left, (_,_,C1,_), (_,_,C2,_)], Score) :-
priority(left,Priority),
Score is (C2 - C1) * Priority.

evaluateMovement([right, (_,_,C1,_), (_,_,C2,_)], Score) :-
priority(right,Priority),
Score is (C2 - C1) * Priority * -1.
```

```
evaluateMovement([down, (_,R1,_,_), (_,R2,_,_)], Score) :-
priority(down,Priority),
Score is (R2 - R1) * Priority * -1.

priority(left,10).
priority(right,10).
priority(down,1).
priority(rotate,1).
```

The 'evaluateMovement/2' predicate, designed to adhere to the requirements of the heuristic function provided by the planner module, computes the difference in coordinates between the current node and the goal node. This computation is adjusted to accommodate the backward search strategy employed by the planner. furthermore when an action would move the tetromino toward the goal, the result of the computation is negated to prioritize such actions when ordering the list of available actions.

Additionally, a multiplication by the priority of the move is performed. Here, the priority is inversely proportional to the associated value; lower priorities correspond to higher values. This approach ensures that moves typically performed earlier in standard scenarios are executed first, thus preventing random action selection when the costs between different actions are similar.

The priority values are shaped around the following typical algorithms when humans play Tetris:

- Align the tetromino to the goal column.
- Rotate it until it's in the correct rotation state.
- Push down the tetromino towards the goal.

However, since the search in TetrisVS is performed backward, the priority values are inverted giving priority to rotation and pushing actions and then to horizontal movement. This optimization works very well in typical scenario and allows to find the proper path quickly, but it could be not optimal when more complex movements are needed. These scenarios, practically impossible in Solo mode, can be intentionally created by human players in VS mode, albeit without necessarily reflecting a proper strategy.

Goal checker predicate:

```
checkGoal(Node,Node).
```

This predicate is trivially true when both terms are identical. In TetrisVs, the planner is invoked with start and goal nodes represented as compound terms '(T1, Y, X, GbList)', where 'T1' represents the tetromino symbol along with its rotation state, 'Y' and 'X' denote the row and column coordinates, respectively, and 'GbList' denotes the game board list through which the tetromino will move.

The planner is invoked using the following predicate that put everything seen in this subsection together.

```
serchPath(Start, Goal, Plan, PlanStory) :-
    findall(A, action(A), Actions),
    Heuristic=evaluateMovement,
    GoalChecker=checkGoal,
    planner(Start, Goal, Actions, Heuristic, Plan, PlanStory, GoalChecker).
```

2.5 Min-Max Algorithm

Utilizing the predicates presented so far, a small Tetris-playing agent could be implemented; however, it would lack the capability to manage VS mode or devise complex plans using subsequent tetrominoes in Solo mode. To address these challenges, a custom version of the min-max algorithm with alpha-beta pruning[16] was developed. Similar to the planner discussed earlier, this component was designed with adaptability and generality in mind, facilitating its potential use in other gaming contexts.

The algorithm was implemented to serve two distinct purposes: as a classic min-max with alpha beta pruning algorithm for zero-sum games and as a look-ahead mechanism for Solo mode. This was achieved by introducing a new perspective player, 'maxmax'. When 'maxmax' is used as starting player instead of 'max', the algorithm maintains the same perspective throughout each layer of the tree returning the move that maximizes the overall utility.

The prolog implementation of the min-max module is the following.

```
:- module(minmax, [minmax/10]).
nextplayer(max,min).
nextplayer(min,max).
nextplayer(maxmax,maxmax).
operator(max,'@>=').
operator(min,'@=<').
operator(maxmax,'@>=').

%minmax, max depth reached
minmax(CurrentNode, _, Heuristic, _, Depth, Depth, _, _, CurrentNodeEvaluated, Player) :-
    evaluate(Player,Heuristic,CurrentNode,CurrentNodeEvaluated),!.

%minmax recursive case
minmax(CurrentNode, NextNodesGenerator, Heuristic, MoveTaker, Level, Depth, Alpha, Beta,
CurrentNodeEvaluated, Player) :-
    call(NextNodesGenerator,Player,Level,CurrentNode,NextNodes),
    minmax1(NextNodes,CurrentNode, NextNodesGenerator, Heuristic, MoveTaker, Level, Depth,
    Alpha, Beta, CurrentNodeEvaluated, Player).

%no successor for the current node
```

```
minmax1([],CurrentNode, _, Heuristic, _, _, _, _, _, CurrentNodeEvaluated, Player) :-
    evaluate(Player,Heuristic,CurrentNode,CurrentNodeEvaluated),!.

%there are successors for the current node
minmax1(NextNodes, CurrentNode, NextNodesGenerator, Heuristic, MoveTaker, Level, Depth,
Alpha, Beta, CurrentNodeEvaluated, Player) :-
    iterativeEval(NextNodes, EvaluatedNodes, NextNodesGenerator, Heuristic, MoveTaker,
    Level, Depth, Alpha, Beta, Player),
    operator(Player,Operator),
    sort(1, Operator, EvaluatedNodes, SortedEvaluatedNodes),
    nth1(1,SortedEvaluatedNodes,BestNode),
    getBest(MoveTaker,Level,CurrentNode,BestNode,CurrentNodeEvaluated).

iterativeEval([CurrentNode|NextNodes], [CurrentNodeEvaluated|NextNodes2],
NextNodesGenerator, Heuristic, MoveTaker, Level, Depth, Alpha, Beta, Player):-
    Level1 is Level + 1,
    nextplayer(Player,NextPlayer),
    minmax(CurrentNode, NextNodesGenerator, Heuristic, MoveTaker, Level1, Depth, Alpha,
    Beta, CurrentNodeEvaluated, NextPlayer),
    updateAlphaBeta(Player, CurrentNodeEvaluated, Alpha, Beta, NextNodes, Alpha1,
    Beta1, NextNodes1),
    iterativeEval(NextNodes1, NextNodes2, NextNodesGenerator, Heuristic, MoveTaker, Level,
    Depth, Alpha1, Beta1, Player).

iterativeEval([], [], _, _, _, _, _, _, _, _).

getBest(_,0,_,BestNode,BestNode) :- !.

getBest(MoveTaker,_,CurrentNode,BestNode,CurrentNodeEvaluated) :-
    nth1(1,BestNode,Evaluation),
    call(MoveTaker,BestNode,Moves),!,
    append([Evaluation,Moves],CurrentNode,CurrentNodeEvaluated),
    !.

evaluate(Player,Heuristic,CurrentNode,CurrentNodeEvaluated) :-
    call(Heuristic,Player,CurrentNode,CurrentNodeEvaluated).

%AlphBeta pruning disabled for maxmax mode
updateAlphaBeta(maxmax,_,Alpha,Beta,NextNodes,Alpha,Beta,NextNodes).

updateAlphaBeta(max,CurrentNodeEvaluated,Alpha,Beta,NextNodes,Alpha1,Beta1,NextNodes1) :-
    nth1(1,CurrentNodeEvaluated,Eval),
    Alpha1 is max(Eval,Alpha),
    Beta1 = Beta,
    pruneNextNodes(Alpha1,Beta1,NextNodes,NextNodes1).
```

```
updateAlphaBeta(min,CurrentNodeEvaluated,Alpha,Beta,NextNodes,Alpha1,Beta1,NextNodes1) :-
    nth1(1,CurrentNodeEvaluated,Eval),
    Beta1 is min(Eval,Beta),
    Alpha1 = Alpha,
    pruneNextNodes(Alpha1,Beta1,NextNodes,NextNodes1).

pruneNextNodes(Alpha1,Beta1,_,NextNodes1) :-
    Beta1 =< Alpha1,
    NextNodes1 = [],
    !.

pruneNextNodes(_,_,NextNodes,NextNodes).
```

The predicate 'minmax/10' is exposed to the outside and takes the following inputs:

- 'CurrentNode': the current node under evaluation within the game tree, where a node is a certain game-state configuration represented as a list.
- 'NextNodesGenerator': a predicate that can generate the subsequent states starting from the one in 'CurrentNode'.
- 'Heuristic': a predicate that computes the static evaluation of a node reflecting its utility.
- 'MoveTaker': a predicate that can manipulate the node structure in order to extract and propagate information useful for the explainability of the algorithm. This predicate is necessary considering that the structure and pertinent information within a node may vary depending on the actual managed game.
- 'Level': the value of the current level of the min-max tree.
- 'Depth': the value of the depth limit reachable by the algorithm.
- 'Alpha' and 'Beta': the current alpha and beta values related to the current node. These values allows to implements the alpha-beta cuts, a technique that allows to save computational time by pruning irrelevant parts of the game tree.
- 'Player': the player associated with the current node. When 'Player' is set to 'max', the predicate operates as a standard min-max algorithm, alternating between maximizing and minimizing players at each level of the game tree. Alternatively, setting 'Player' to 'maxmax' enables the algorithm to work with a simple look-ahead approach.

And give as output:

- 'CurrentNodeEvaluated': if 'Level' is greater than 0 this variable contains the same structure as 'CurrentNode' with an additional score attached at the head of the list. This score represents the evaluation of the current node. The score is computed by the 'Heuristic' predicates in the case of leaf node, or inherited when dealing with intermediate nodes.

  If 'Level' equals 0, 'CurrentNodeEvaluated' holds the best successor node identified during the search, which serves as the optimal move to execute.

The predicate 'minmax/10' search the best move for the current player by sexecuting the following steps:

1. Checks if the maximum depth has been reached by trying to unify 'Level' and 'Depth' at the same value, then perform the static evaluation on the current node and unify it in 'CurrentNodeEvaluated'.

2. If the previous condition doesn't hold, the set of successor nodes is generated, then 'minmax1/11' is called.

3. When the list of successors node is empty, 'minmax1/11' just perform the static evaluation on the current node and unify it in 'CurrentNodeEvaluated'.

4. Otherwise, 'minmax1/11' calls 'iterativeEval/10'.

   This predicate increments the current level, switch the player and calls 'minmax/10' on the first node in the 'NextNodes' list. Here is where the recursive behavior of the min-max algorithm take place.

   When the recursive execution of 'minmax/10' ends, 'iterativeEval/10' updates the alpha and beta values calling 'updateAlphaBeta/8' accordingly to the player holding the node. This predicate, when beta is less or equal to alpha, will prune part of the tree by empty the 'NextNodes' list and so terminating the execution of 'iterativeEval/10' for that node. Otherwise 'iterativeEval/10' is called again on the remaining part of the list 'NextNodes'.

   When 'iterativeEval/10' ends, a new list of evaluated nodes is produced in 'EvaluatedNodes' and the execution flow goes back to 'minmax1/11'.

5. 'minmax1/11' orders the resulting list in 'EvaluatedNodes' according to the node-holding player perspective. When it is 'max' or 'maxmax' the results are ordered in descending order, this because they want to maximize their utility and so are interested in the greater node containing the best move. When it is 'min' the result are ordered in ascending order because it wants to minimize the 'max' utility, and so it is interested in the worst available move.

6. If the level is greater than zero, the first element in the 'SortedEvaluatedNodes' is taken into account, its score is attached to the current node along with the information regarding the chain of move until the current

node. If the level is zero the best node is simply returned unifying it in 'CurrentNodeEvaluated'. This node will contain the move that will be executed by the AI agent.

The min-max module is used by TetrisVS providing the following lists and predicates.

StartingNode list:

```
[T,GbL,0]
```

This list is passed as the initial value for 'CurrentNode' when 'minmax/10' is called and represents the root node, from which the best available move needs to be determined. 'T' represents the list of available tetrominoes, 'GbL' represents the current game board, and 0 represents the number of rows actually cleared. This last value is passed because, in the top-down flow of the nodes, it is necessary to compute the final score of a leaf node.

NextNodesGenerator predicates:

```
%easy mode: every row-clearing move stops the branch exploration
nextNodes(Player,_,Node,[]) :-
    easyMode,
    Player \= 'maxmax',
    nth1(3, Node, ClRow),
    ClRow > 0.

%hard mode: only the opponents row-clearing move stops the branch exploration,
%the AI player will continue the branch exploration until the end.
nextNodes(Player,_,Node,[]) :-
    \+easyMode,
    Player = 'max',
    nth1(3, Node, ClRow),
    ClRow > 0.

nextNodes(Player,Level,Node,NextNodes) :-
    nth1(1, Node, Tetrominoes),
    nth1(2, Node, GbL),
    nth0(Level, Tetrominoes, T),
    nth1(3, Node, ClRow),
    nextNodes1(Player,Tetrominoes,GbL,T,ClRow, NextNodes).

nextNodes1(_,Tetrominoes,GbL,T, _, NextNodes) :-
    findPossibleGoals(T,L,GbL),
    callPlacePiece(Tetrominoes,GbL,L,NextNodes).

callPlacePiece(_,_,[],[]).
```

```
callPlacePiece(Tetrominoes,GbList,[(T,R,C)|Taill],
[[Tetrominoes,GbListPost,ClRow,(T,R,C)]|Tail2]):-
    catch(call_with_time_limit(0.5, checkElegibility(T,R,C,GbList)),
    time_limit_exceeded,fail),
    placePiece(T,R,C,GbList,GbListPost,ClRow),
    callPlacePiece(Tetrominoes,GbList,Taill,Tail2),
    !.

callPlacePiece(Tetrominoes,GbList,[_|Taill],Tail2):-
    callPlacePiece(Tetrominoes,GbList,Taill,Tail2).
```

The predicate 'nextNodes/4' generates the next successors node of the node in the variable 'Node'.

There is an essential clarification to make before proceeding further with the dissertation.

The win condition of TetrisVs in Vs mode is to make more points than the opponent. However, given a game state, it is possible only to foresee up to the current player's next tetromino. This because the subsequent tetrominoes are generated randomly being the game stochastic in its nature. Because of this limitation, is impossible to use the min-max algorithm for reaching the real win condition of the game, moreover, even if it was possible it would be really computationally expensive to achieve due to the combinatorial explosion of states.

To overcome this limitation a fictitious win condition was defined. In this modified scenario, a player is considered to have achieved victory when she clears one or more rows on the game board. By introducing this artificial win condition, the AI can focus its evaluation on the available game states and effectively assess strategies aimed at clearing rows, which indirectly contribute to scoring points and gaining an advantage over the opponent. Without this adjustment, the agent would be unable to properly recognize situations where the opponent could potentially clear rows, leading to sub-optimal decision-making and impaired performance.

In order to implement this trick, the predicate 'nextNodes/4', only when the player are 'min' or 'max', can prune the nodes generation when the number of cleared rows in the current node is greater than zero. This forces the min-max algorithm to perform the static evaluation on the current node without proceeding further. This also opened to the possibility of implement two levels of AI strength, the easy and the hard mode.

In easy mode, node generation is halted regardless of the current player's perspective. Thus, even when the 'min' player is evaluating its current state which is the result of a 'max' move, the process stops if a row-clearing move is encountered. This approach may lead to non-quiescent states, where the opponent is presented with opportunities to clear rows in the next turn.

Conversely, in hard mode, the pruning technique is only applied when the current player is 'max', which game states are the result of a 'min' move. In the

other case, the algorithm continues to expand the game tree until reaching leaf nodes. This allows the 'max' player to anticipate situations where a row-clearing move would potentially benefit the opponent, thereby avoiding such scenarios.

The strength of the AI is selected by the client asserting eventually the 'easyMode/0' fact.

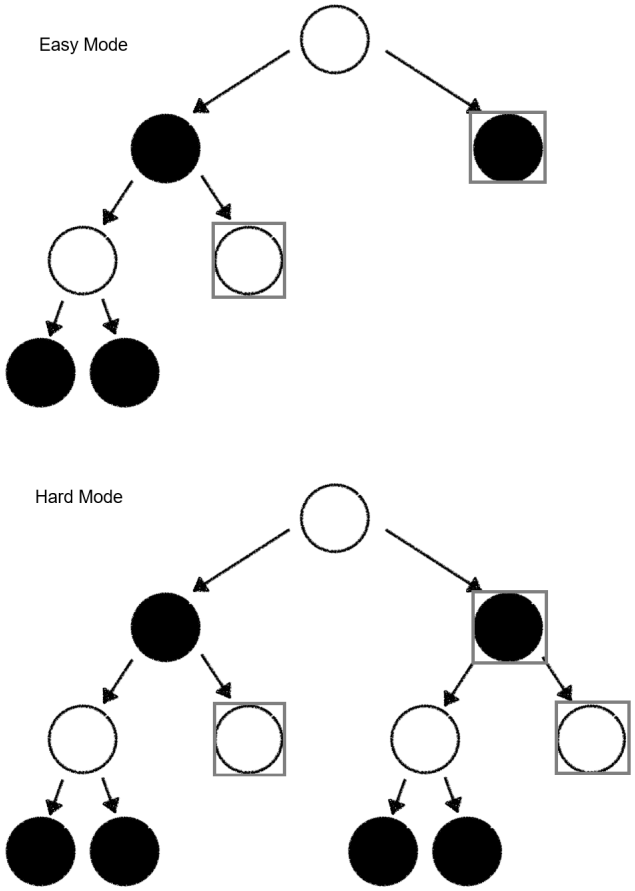

Figure 4: A graphical representation of the same game tree, approached using min-max in easy mode and min-max in hard mode. The white represents "max" nodes and the black represents "min" node. The node in the grey square are nodes where a row-clearing move has been executed. In easy mode "max" generate a row-clearing move and then prune the tree to force evaluation. In hard mode the same node is furtherly explored discovering a "min" move that would clear some row, and then pruning it to force evaluation. By this way the min-max will avoid to move the AI player towards that case.

In all other cases, the purpose of 'nextNodes/4' is to generate the succes-

sors of the current node. This process involves calling 'findPossibleGoals/3' to identify potential goals and then, iteratively for each eligible goal, invoking 'placePiece/6' to create the list of successors to store in 'NextNodes'. It's essential to verify the eligibility of a goal since 'findPossibleGoals/3' doesn't inherently ensure that the proposed goal is reachable by the tetromino. This check is achieved using the previously introduced planner[8].

Heuristic predicates:

```
evaluateNode(Player,[Tetrominoes,GbL,ClRow,(T,R,C)], [S,Tetrominoes,GbL,
(AggHeight,ClRow,Holes,Bump,Ent),(T,R,C)]) :-
    gameBoardScore(Player,GbL,AggHeight,ClRow,Holes,Bump,Ent,S),
    !.
```

The predicate 'evaluateNode/3' perform the evaluation of the node calling 'gameBoardScore/8' generating a new node containing the computed score at the head of the node's list, along with the score's component. These components are represent as a compound term that replace the number of cleared rows in the structure of the node.

The predicate 'gameBoardScore/8' is the expanded version of the one already discussed in the Game Board Evaluation Predicates subsection. It adds a new argument in input that contain the player requesting the evaluation. This predicate is so defined.

```
gameBoardScore('max',GbList,AggregateHeight,RowsCleared,Holes,Bumpiness,SumEnt,Score) :-
    RowsCleared > 0,
    gameBoardScore(GbList,AggregateHeight,0,Holes,Bumpiness,SumEnt,Score1),
    Score is -1*1000*RowsCleared+Score1,
    !.

gameBoardScore('min',GbList,AggregateHeight,RowsCleared,Holes,Bumpiness,SumEnt,Score) :-
    RowsCleared > 0,
    gameBoardScore(GbList,AggregateHeight,0,Holes,Bumpiness,SumEnt,Score1),
    Score is 1000*RowsCleared+Score1,
    !.

gameBoardScore(_,GbList,AggregateHeight,RowsCleared,Holes,Bumpiness,SumEnt,Score) :-
    gameBoardScore(GbList,AggregateHeight,RowsCleared,Holes,Bumpiness,SumEnt,Score).
```

If called by 'max' or 'min' and 'RowsCleared' is greater than zero, it recall 'gameBoardScore/7' adjusting the score adding a value of $1000 * RowsCleared$

[8] Generating eligible goals without using the path-finding algorithm was challenging. Initial attempts were unsuccessful, leading to a switch to the current solution. The 'checkElegibility/4' predicate imposes a 0.5 time constraint to abort lengthy computations. While this may occasionally discard good moves in rare scenarios, it's unlikely unless intentionally done by human players.

[9]. When the caller is 'max', this value is multiplied by -1. This adjustment is necessary because at the 'max' level, the algorithm is actually evaluating states that are the result of a 'min' node interested in minimizing the utility. Conversely, when the caller is 'min', the opposite adjustment is applied. This adjustment ensure that all the row-clearing move are prioritized accordingly with the player perspective making possible to implement the fictitious win condition previously described.

In all the other cases 'gameBoardScore/7' is recalled without any adjustment in value.

MoveTaker predicates:

```
%base case: when we collect the value from a leaf
takeMove(Node,[Move,ScoreComponent]):-
    length(Node,5),
    nth1(4,Node,ScoreComponent),
    nth1(5,Node,Move).

%recursive case: when we collect the value from non-leaf node
takeMove(Node,[Move|CollectedMoves]):-
    nth1(2,Node,CollectedMoves),
    nth1(6,Node,Move).
```

The 'takeMove/2' predicate serves to extract the move associated with a node selected as the best, facilitating the reconstruction of the chain of moves that led to the wanted game state. This functionality was implemented to enhance the explainability of the algorithm. When dealing with a leaf node of length 5, the predicate simply returns the move along with its score components. However, when dealing with a non-leaf node where part of the chain has already been added, the move of that node is extracted from the sixth position, without including the score components (which are nonexistent for intermediate nodes).

The min-max module is invoked by TetrisVs using the following predicate that put everything seen in this subsection together.

```
callMinMax(GbL, Player, BestNode) :-
    tetrominoes(T),
    length(T,Depth),
    StartingNode = [T,GbL,0],
    NextNodesGenerator = nextNodes,
    Heuristic = evaluateNode,
    MoveTaker = takeMove,
    minmax(StartingNode, NextNodesGenerator, Heuristic, MoveTaker, 0, Depth, -inf,
    +inf, BestNode, Player).
```

[9] In this scenario, 'gameBoardScore/7' doesn't directly compute the score component related to the cleared rows. This is done to prioritize the custom adjustment made for the Vs mode, allowing it to have more relevance in determining the final score.

It is possible to notice that 'Level' is set to 0 as first value and 'Alpha' and 'Beta' are set to $-\infty$ and $+\infty$ respectively, as required by the alpha-beta pruning algorithm. These values are set basically in the same way regardless the actually implemented game[10].

To conclude this subsection, a brief recap of the structure of a min-max node and its evolution is provided:

- '[Tetrominoes, GbL, 0]' represents the starting node, as discussed earlier.
- '[Tetrominoes, GbL, RowsCleared, Move]' is the structure of a node generated by 'nextNodes/4', with the move that generated the node added at the tail.
- '[Eval, Tetrominoes, GbL, (AggHeight, RowsCleared, Holes, Bump, Ent), Move]', is the structure of a node generated by 'evaluateNode/3', with the evaluation score added at the head and the compounded term replacing the RowsCleared term.
- '[Eval, MoveChain, Tetrominoes, GbL, (AggHeight, RowsCleared, Holes, Bump, Ent), Move]' is the structure of a node selected as the best node at a certain level of the game-states tree, with the 'MoveChain' list added after the score. The number of moves in this list will grow up to the actual depth of the tree when reaching the root. This is the structure of the actually returned node.

2.6 Generating the Best Move

Using the so far defined predicate is now finally possible to define a predicate that, given a player, the asserted game board and the asserted set of tetrominoes, produces the list of moves to execute in order to reach the configuration that maximize the player's utility.

```
getPathOfBestMove(Player,Plan) :-
    getStartGbL(GbL),
    tetrominoes([T|_]),
    firstShape(T,T1),
    tetrominoSpawnX(X),
    tetrominoSpawnY(Y),!,
    placePiece(T1,Y,X,GbL,_,_),  condition if failed
    callMinMax(GbL,Player,BestNode),
    Start = (T1,Y,X,GbL),
    last(BestNode,(Tg,Rg,Cg)),
    Goal = (Tg,Rg,Cg,GbL),
    serchPath(Goal, Start, RevPlan, PathStory),
    reverse(RevPlan,Plan),
```

[10] Another example of using the min-max module will be explored in the section Appendix B: Tris.

```
assertGbL(BestNode),
assertExplanation(BestNode,Tg,Rg,Cg,PathStory),
!.
```

The predicate 'getPathOfBestMove/2' takes as input the player, 'max' if the game is in Vs mode or 'maxmax' for the Solo mode, and provides the list of moves in the 'Plan' variable. It essentially executes 'callMinMax/3' using the available information and then calls 'serchPath/4' to compute the list of moves. Notably, the goal and the start are passed as the first and second arguments, respectively, so the planner will use the goal as the starting node and the start as the goal node, consistent with the backward searching strategy described earlier. Finally, the produced plan is reversed and unified with the variable 'Plan', and the new game board list is asserted replacing the old one, along with the information useful for explainability.

2.7 SWI-Prolog Web Server Implementation

The described agent capabilities are offered through a set of REST API endpoints to the JavaScript client that implements the interactive part of the game. These endpoint has been implemented using the 'http' library offered by SWI-Prolog and exposed trough the 7777 port. The available endpoints are the followings:

- */home*: this endpoint is used to display an HTML page showing the current game board and available tetrominoes. It's primarily used for debugging purposes to ensure that the agent's internal representation of the game board matches the actual game state.
- */retcell*: used to clear the agent's representation of the game board, removing all information about the current state and the explainability data related to the last move executed.
- */put/Tr/R/C* : this endpoint is used to inform the agent about the move made by the human player in Vs mode. It calls the 'placePiece/3' predicate to place tetromino 'Tr' (tetromino in a given rotation state) at coordinates 'R' and 'C' on the game board. The new game board is then asserted, replacing the old one.
- */resetstart*: used to clear information about the available tetrominoes.
- */start/T*: asserts tetromino 'T' as available. It adds 'T' to the list of available tetrominoes held by the 'tetrominoes/1' fact.
- */easyMode*: asserts the 'easyMode/0' fact, allowing the agent to use the min-max algorithm in easy mode.
- */hardMode*: retracts the 'easyMode/0' fact, allowing the agent to use the min-max algorithm in hard mode.

- */path/Player*: used to obtain the path towards the best move given as list of moves. It calls 'getPathOfBestMove(Player,Plan)'.
- */explainMove*: retrieves the move chain contained in the 'explanation/3' fact used to explain the rationale behind a decision made by the min-max algorithm.
- */explainPath*: this endpoint displays an HTML page that presents the path story contained in the 'explanation/3' fact, showing the evolution of the searching process performed by the path-finding algorithm. Additionally, it provides an explanation text to further clarify the process.
- */explainHeuristic*: displays an HTML page showing how much each component has contributed to the evaluation of the heuristic of the selected best move using the information held by the 'explanation/3' fact. It's important to note that these information are not relative to the actually executed move, but to the foreseen move that the min-max algorithm wants to reach. This is because the game board evaluation is performed only for terminal nodes.

The JavaScript client interacts with these endpoints to communicate with the agent. The */retcell* and */resetstart* endpoints are called at the beginning of each new game, irrespective of the game mode, to clear the agent's memory.

In Solo mode, the client communicates with the server for each available i tetromino by calling the */start/Ti* endpoint. This transmits the new tetrominoes array to the agent. Subsequently, the client invokes the */path/maxmax* endpoint to retrieve the path to the best move and the */explainMove* endpoint to obtain information related to the explainability of the move.

In Vs mode, when it's the human player's turn, the client calls the *put/Tr/R/C* endpoint after her move to communicate the position of the new tetromino and update the agent's internal board representation.

During the AI's turn, the same endpoints used in Solo mode are called. However, when determining the AI's move, the */path/Player* endpoint can be invoked in two different ways. If the height of the highest column is less than fourteen, the endpoint is called with *max* as the 'Player'. Otherwise, *maxmax* is used. In the former case, the agent employs the min-max algorithm in the regular competitive mode. In the latter case, known as the "Emergency Mode", the agent is tasked with finding the best move to clear lines and avoid a game over situation.

2.8 Explainability of the Agent's Decision

One key feature of the current AI agent is its ability to explain its decisions. This capability is enabled by several design decisions that contribute to this result. The 'explanation/3' fact holds the move chain produced by the min-max algorithm, the path story produced by the path-finding algorithm, and the score component related to the best foreseen move each time a new move is

computed. However, proper management of explainability is also essential in the GUI design to help users interpret the provided information correctly. In TetrisVs, these useful pieces of information are extracted by a set of endpoints, as discussed earlier, and then displayed by the client in the console log in an interpretable way.

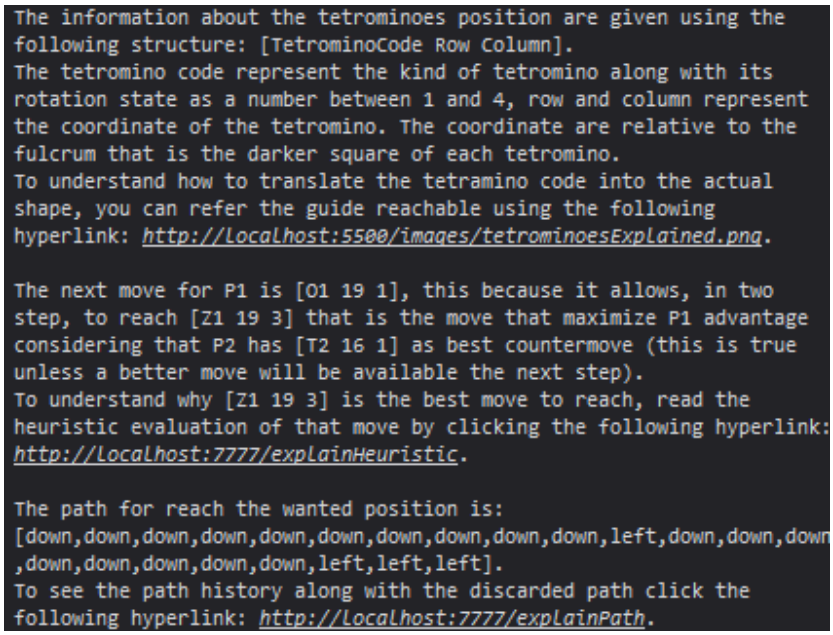

Figure 5: An example of the text produced by the client in order to explain the AI agent move.

Additionally, the explanation provides access to the image 1 that assists the user in understanding how the coordinates are managed. Furthermore, the graphical representation of the game board includes row and column numbers that can be used as references.

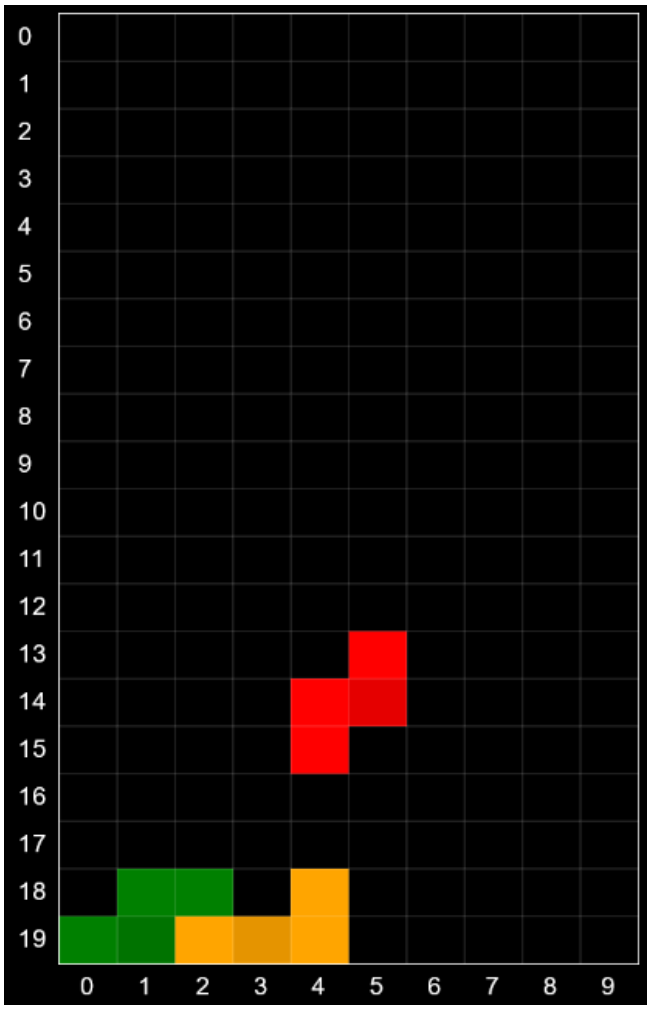

Figure 6: The game board is displayed with rows and columns numbered for reference. Additionally, each tetromino's pivot point is highlighted with a darker color. This aids the user in navigating the coordinate system and comprehending the explanations provided.

3 Performances, Current Limitations and Future Developments

The presented agent has demonstrated its ability to efficiently manage the game of TetrisVs in both of its game modes. To evaluate its capabilities in Solo mode 10 simulated runs was executed achieving the following results:

Run Number	Number of Cleared Rows
1	859
2	2660
3	1187
4	2922
5	2497
6	2390
7	1111
8	1127
9	1675
10	1908

Table 1: Number of cleared rows in Solo mode for each run

The agent is capable of reaching an decent number of cleared lines with an average of 1833,60 over ten runs. This result is good if compared to standard human performances as shown in figure 7 but is not comparable to the results discussed in Algorta et al. work[6] where millions of rows has been reported to be cleared. In order to achieve better results a parameter optimization for each component of the heuristic function should be applied as in the already cited Lee's[13] work. Nonetheless it can be considered a satisfactory result considering that the heuristic is not so sophisticated. Another aspect that need to be taken into account is that TetrisVs uses as random generator used to draw a new tetromino the simple random function built in JavaScript, when usually, in other Tetris version, more complex random generators are used in order to avoid sequences of pieces that surely leads to loose the game, like a long sequence of Z and S tetrominoes[17]. Performances of the current AI agent should be evaluated also taking into account such different tetrominoes generator.

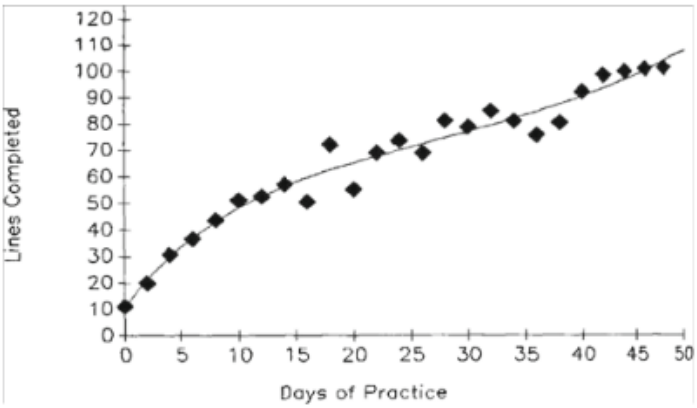

Figure 7: Growth curve of players' average lines cleared as practice days increased. Reprinted from Haier et al. [18].

In Vs mode the AI agent is capable of operating as expected, resulting in a strong opponents for the human player. When operating in hard mode, the AI agent never presents the opponents with the opportunity to clear rows. Conversely, when operating in easy mode, it occasionally allows the opponents to clear rows. Of course regardless the difficulty level, when the game reaches the "Emergency Mode", the human player can exploit this temporary window in order to accumulate more points than the AI agent and eventually beat it. This design choice aligns with the objective of the Vs mode, which aims to prolong gameplay and prevent premature game over scenarios that would result in both the player and the AI losing the game.

One of the primary limitations encountered in implementing the agent for Vs mode is related to the heuristic evaluation of opponents' moves. The explanations provided by the agent sometimes suggest countermoves that may seem unexpected, such as moves that create several "weak" holes instead of fewer, but more effective, "proper" ones. This discrepancy derives from the already discussed absence of the concept of "proper" holes in the current implementation.

The primary limitation of the agent is its slowness, which results in the inability to manage real-time gameplay that has led to the necessity of temporarily pause the game each time it calls the AI in order to wait for its response. In Solo mode, it can generate a plan within a maximum of two seconds, sufficient for real-time interaction, especially in early levels. However, in Vs mode, move generation can take up to ten seconds in complex scenarios, such as chains of three or four rotation pieces. This prevents the agent from effectively engaging in real-time Vs mode gameplay, creating a distinctly different experience compared to playing against a human opponent who must adapt to the game's increasing speed and reason in fraction of seconds.

To address this limitation, research could be conducted into more efficient move generation algorithms that eliminate the need for eligibility checks. Additionally, key components of the agent, such as the heuristic evaluator, could be implemented in low-level languages like C to significantly enhance processing speed, enabling the agent to react quickly and facilitate real-time gameplay. Moreover, employing a low-level language could expedite the checks associated with evaluating "proper" holes, ensuring that they are performed quickly enough to be incorporated into the heuristic. This would help overcome the limitations deriving from their absence in the current implementation. Another way for enhancement could involve adopting a different algorithm for deciding the best move in Vs mode. A viable alternative to min-max is the Monte-Carlo tree search, which suffers less from game-tree branch explosion. This approach involves running several simulations based on random moves, deepening the exploration of the most promising move, and finally selecting the best-found outcome as a result. However, due to its stochastic nature, it could affect the explainability of the selected move, providing a more opaque rationale compared to the deterministic approach of min-max. While Monte Carlo tree search offers the potential for more efficient decision-making by focusing on promising regions of the game tree, it may lack the transparency of min-max in explaining its decisions. Balancing between the efficiency of Monte Carlo tree search and the transparency of decision-making remains a key consideration in implementing AI agents for complex games.

4 Conclusions

The presented work has demonstrated how Prolog can be leveraged as a key tool for the development of effective AI agents capable of handling non-trivial games like TetrisVs while providing clear rationales behind their decisions. Whereas there is still ample room for improvement, these initial results are promising enough to justify further exploration in future game-playing AI projects. Additionally, the two modules presented for path-finding and min-max move selection are easily adaptable to other gaming scenarios. In Sections Appendix A: Snake and Appendix B: Tris, two simpler games utilizing these modules are presented as evidence of their flexibility. The modular design of the presented components not only demonstrates their versatility across different gaming scenarios but also hints at the potential for broader applications beyond gaming.

Appendix A: Snake

Snake is a classic video-game in which players control a snake navigating a grid-based world. The objective is to guide the snake to consume food while avoiding collisions with walls or its own body that increase in length for each food item consumed.

In this simple presented AI-implementation of the game, the planner module

is used to allows the snake to move automatically towards the food. In this section only the predicates pertaining the use of the path-finding planner are discussed, the full source code can be easily understood by reading it in the 'snake.pl' file.

The planner module is called by passing the following predicate.

```
%Actions
action(moveUp).
action(moveLeft).
action(moveDown).
action(moveRight).

moveDown(S,Sp) :-
    nth0(0,S,[R,C]),
    R1 is R+1,
    gameBoardH(H),
    R1<H,
    \+member([R1,C],S),
    append([[R1,C]],S,Sp1),
    withoutLast(Sp1,Sp).

moveUp(S,Sp) :-
    nth0(0,S,[R,C]),
    R1 is R-1,
    R1>=0,
    \+member([R1,C],S),
    append([[R1,C]],S,Sp1),
    withoutLast(Sp1,Sp).

moveLeft(S,Sp) :-
    nth0(0,S,[R,C]),
    C1 is C-1,
    C1>=0,
    \+member([R,C1],S),
    append([[R,C1]],S,Sp1),
    withoutLast(Sp1,Sp).

moveRight(S,Sp) :-
    nth0(0,S,[R,C]),
    C1 is C+1,
    gameBoardW(W),
    C1<W,
    \+member([R,C1],S),
    append([[R,C1]],S,Sp1),
    withoutLast(Sp1,Sp).
```

```
%Evaluation
evaluateMovement([moveUp, S, F], Score) :-
    nth0(0,S,[R1,_]),
    nth0(0,F,[R2,_]),
Score is (R2 - R1) * -1.

evaluateMovement([moveDown, S, F], Score) :-
    nth0(0,S,[R1,_]),
    nth0(0,F,[R2,_]),
Score is (R2 - R1).

evaluateMovement([moveRight, S, F], Score) :-
    nth0(0,S,[_,C1]),
    nth0(0,F,[_,C2]),
Score is (C2 - C1).

evaluateMovement([moveLeft, S, F], Score) :-
    nth0(0,S,[_,C1]),
    nth0(0,F,[_,C2]),
Score is (C2 - C1) * -1.

%Goal checker
checkGoal([H|_],[F|_]) :-
    H == F.

%Calling the planner module
serchPath(Start, Goal, Plan, PlanStory) :-
    findall(A, action(A), Actions),
    Heuristic = evaluateMovement,
    GoalChecker = checkGoal,
    planner(Start, Goal, Actions, Heuristic, Plan, PlanStory, GoalChecker).
```

Where 'moveDown/2', 'moveUp/2', 'moveLeft/2' and 'moveRight/2' all take a list representing the snake and give as result the new list representing the snake body in the new position. The 'evaluateMovement/2' takes as input a triple in a list, as the one seen in TetrisVs, containing the evaluated move, the list representing the snake body, and the position of the food. The evaluation is performed comparing the position of the head of the snake with the position of the food preferring moves that minimize the distance. The 'checkGoal/2' predicate simply compare if the head of the snake is in the same position of the food. In this case we are comparing nodes with different structure, confirming the capability of the planner to adapt to different game's needs.

The current agent successfully navigates towards the food autonomously, following the shortest path. However, once the snake grows longer, the planner

may inadvertently trap the snake within its own body. This occurs because the planner does not prioritize leaving enough space for the snake to maneuver freely throughout the grid.

One potential solution to this limitation involves generating two food items simultaneously and devising a plan that allows the snake to reach both. Once reached a food item, a new one is generated, and the plan is recalculated with the assumption that the first food item remains reachable. However, the planner module is currently unable to handle multiple goals simultaneously, necessitating further development to enable this functionality.

Appendix B: Tris

Tris, also known as Tic-Tac-Toe, is a classic game where two opponents play on a 3x3 grid by placing at each turn the own symbol, usually an 'X' or an 'O', on the board. The player who is capable of aligning three occurrence of its own symbol on the board, horizontally, vertically or diagonally, wins the game.

In the AI field this game is a standard benchmark for assesting the quality of a game-playing algorithm. Given it's simplicity, implementing this game can easily tell if the algorithm works well or not before diving in the application of the algorithm in more complex games.

In the context of this project an AI agent capable of playing Tris has been implemented in order to test the presented min-max module and demonstrate its adaptability for other games. In this section only the predicates pertaining the use of the min-max module, the full source code can be easily understood by reading it in the 'tris.pl' file.

The min-max module is called by passing the following predicates.

```
%%NextNodesGenerator
availableMoves(G,L) :-
    findall(I, (nth0(I, G, e)), L),
    !.

availableMoves(_,[]).

%generate next nodes
nextNodes(_, _, [G,P,_], [[Gp,P2,[Ly,P]]|NextNodes]) :-
    next(P,P2),
    \+winningGB(P2,G),
    availableMoves(G,[Ly|L]),
    !,
    putSymbol(Ly,P,G,Gp),
    nextNodes([G,P,_], L, NextNodes).

nextNodes(_, _, [G,_,_], []) :-
    availableMoves(G,[]).
```

```
nextNodes([G,P,_], [Ly|L], [[Gp,P2,[Ly,P]]|NextNodes]) :-
    next(P,P2),
    putSymbol(Ly,P,G,Gp),
    nextNodes([G,P,_], L, NextNodes).

nextNodes(_, [], []).

%%Heuristic
evaluateNode(Player,[G,P,M],[S,G,P,M]) :-
    winningGB(_,G),
    evaluateNodeValue(Player,S),
    !.

evaluateNode(_,[G,P,M],[S,G,P,M]) :-
    S is 0.

evaluateNodeValue(min,1).
evaluateNodeValue(max,-1).

%allows to remember the move chain
%base case: when we collect the value from a leaf
takeMove([_,_,_,Move],[Move]).

%recursive case: when we collect the value from non-leaf node
takeMove([_,CollectedMoves,_,_,Move],[Move|CollectedMoves]).
%

callMinMax(BestNode) :-
    gB(G),
    turn(P),
    StartingNode = [G,P,[]],
    NextNodesGenerator = nextNodes,
    Heuristic = evaluateNode,
    MoveTaker = takeMove,
    minmax(StartingNode, NextNodesGenerator, Heuristic, MoveTaker, 0, 9, -inf, +inf,
    BestNode, max).
```

Te predicate 'nextNodes/4' is the one called by the min-max module. It first check if the actual grid configuration is not a winning state, then generates all the available empty cells by checking the presence of the 'e' symbol on the board and returning the coordinates as available move, then it calls 'nextNodes/3' on this list of coordinates in order to generate the actual grid configuration after the execution of each available move for that player. If it is a winning state 'nextNodes/4' will unify the list of next nodes with the empty list, resulting in making the current node a leaf node.

The evaluation function is performed by the 'evaluateNode/3' predicate and is called only for winning state, since they are computable and fast to generate, so there aren't static evaluation function performed on intermediate game states. The evaluation function simply check if the current game sate is a winning state and then associate a value of 1 for moves deriving from the 'max' player (so a 'min' node) and -1 otherwise.

The 'takeMove/2' simply takes the executed move from the end of the node list and put it in the move-chain.

The resulting AI-agent is fully capable of playing Tris without loosing any game and achieving victory when the occasion is presented. Furthermore it is fast, given the small search space of the game. However, a notable drawback is the absence of a randomizer, which would introduce variability into the agent's initial moves and make it less predictable. This is particularly evident when calling the 'aiVsAi/0' predicate that run a game between two AI opponents. Currently, the agent always proposes the same optimal response for a given configuration, even when other equivalent actions may be available. Implementing randomization for game states evaluated with the same score would enhance the agent's adaptability and make it more enjoyable to challenge.

References

[1] T. Wiki. Tengen tetris for nes. [Online]. Available: https://tetris.wiki/Tetris_(NES,_Tengen) 1

[2] J. S. Baird, "Tetris," 2023. 1

[3] H. J. Hoogeboom and W. A. Kosters, "The theory of tetris," 2005. 1

[4] S. Deb, "Boy, 13, is believed to be the first to 'beat' tetris," *The New York Times*. [Online]. Available: https://www.nytimes.com/2024/01/03/arts/tetris-beat-blue-scuti.html 1

[5] F. Cadalanu, "I limiti del tetris sono ancora inesplorati," *Ultimo Uomo*. [Online]. Available: https://www.ultimouomo.com/limiti-tetris-competitivo-come-funziona-record-scuti/ 1

[6] S. Algorta and Özgür Şimşek, "The game of tetris in machine learning," 2019. 2, 31

[7] G. W. Record. Most ported videogame. [Online]. Available: https://www.guinnessworldrecords.com/world-records/most-ported-computer-game/ 2

[8] T. Wiki. Nintendo rotation system. [Online]. Available: https://tetris.wiki/Nintendo_Rotation_System 2

[9] Wikipedia. Deep reinforcement learning. [Online]. Available: https://en.wikipedia.org/wiki/Deep_reinforcement_learning 5

[10] G. Kohs. Alphago - the movie. [Online]. Available: https://www.youtube.com/watch?v=WXuK6gekU1Y 5

[11] O. Roeder, *Seven Games: A Human History*, 2022. 5

[12] Wikipedia. Entropy (information theory). [Online]. Available: https://en.wikipedia.org/wiki/Entropy_(information_theory) 11

[13] Y. Lee. Tetris ai – the (near) perfect bot. [Online]. Available: https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/ 12, 31

[14] Wikipedia. Depth-first search. [Online]. Available: https://en.wikipedia.org/wiki/Depth-first_search 12

[15] HardDrop. T-spin guide. [Online]. Available: https://harddrop.com/wiki/T-Spin_Guide 15

[16] Wikipedia. Alpha-beta pruning. [Online]. Available: https://en.wikipedia.org/wiki/Alpha-beta_pruning 17

[17] T. Fandom. Random generator. [Online]. Available: https://tetris.fandom.com/wiki/Random_Generator 31

[18] R. J. Haier, B. V. Siegel, A. MacLachlan, E. Soderling, S. Lottenberg, and M. S. Buchsbaum, "Regional glucose metabolic changes after learning a complex visuospatial/motor task: a positron emission tomographic study," *Brain Research*, vol. 570, no. 1, pp. 134–143, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/000689939290573R 32