ColorShapeLinks AI

ELMEHDI DIAB. DOINA LOGOFATU*
Electrical Engineering department
Frankfurt University of Applied Science
el.diab@stud.fra-uas.de
†Faculty of Computer Science and Engineering
Frankfurt University of Applied Sciences
Nibelungenplatz 1, 60318 Frankfurt am Main, Germany
logofatu@fb2.fra-uas.de

***Abstract*—SimpAI, an AI agent developed for the ColorShapeLinks contest, based on a version of the simplified board game. We introduce SimpAI. A very efficient parallel minimax search with a heuristic function consisting of many partial heuristics which have been improved with an evolving algorithm in balance. SimpAI was the runner-up in the toughest session, requiring a solid adaptability agent for the AI.**

I. Introduction

In this paper, we present SimpAI, an artificial intelligence (AI) agent created for the ColorShapeLinks board game competition [1]. This competition is based on an arbitrarily sized version of the Simplexity board game [1]. In this game, it's a grid positioned vertically, with gravity dropping pieces. The first person to put the like pieces on a sequence wins in a similar way to Connect-4. Color, white or red and form, round or square is specified. Pieces. The first player is won by round or white, while the second player is won by red or quadratic pieces. Shape takes priority over color as a winning state, and while players only play with pieces of their colour, both shapes can participate, and the game is more fascinating and complicated.

In a standard Simplexity game, the board has six rows and seven columns, and victory can be achieved with a sequence of four pieces of the same shape or color, either vertically, horizontally or diagonally. Fig. 1 shows the possible [1]

victory conditions for both players in a standard Simplexity game, i.e., either by color or by shape, with the latter having priority as shown in Fig. 1b and Fig. 1d.

The SimpAI agent was developed as a learning and exploration project in the context of an AI course unit at Lusófona University's Bachelor in Videogames degree [1]. The search board uses a traditional Minimax search technique to look for the best answers, with a number of optimisation options for as much depth as feasible. The search is driven by five heuristics, whose weights were improved by means of a developmental method. The agent finished second in the Unknown Track, where the board settings and time to play were only disclosed after the competitive deadline.

This paper is organized as follows. In Section 2, A Starting with Problem description, as well as the rules of ColorShapeLinks competition works. In Section 3, discusses related work on in board game AI and how ColorShapeLinks in its . In Section 4 we describe the proposed solution obtained using this MiniMax approach and Multithreading (Tasks), how the agent implemented in the competition track. Also a discussion of experimental results and how it perform. Section 5 closes the paper, discussing potential improvements and offering some conclusions.

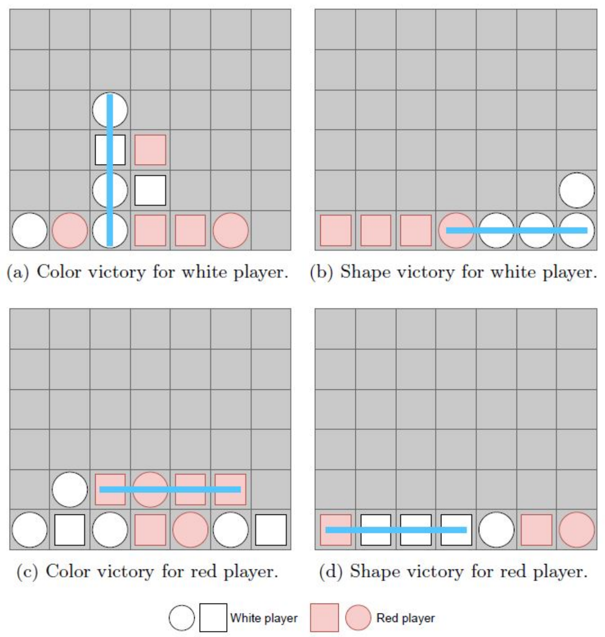

(a) Color victory for white player.
(b) Shape victory for white player.
(c) Color victory for red player.
(d) Shape victory for red player.
White player
Red player

Fig. 1: Victory conditions for the Simplexity board game.

II. Problem Description

ColorShapeLinks was accepted as an official AI competition at the IEEE CoG 2020 conference, and was funded with a prize money of 500 USD for the winner of each track. The competition ran on two distinct tracks:[2] [3]

1. The Base Track, which used standard Simplexity rules with a time limit of 0.2 seconds per move. Only one processor core was available for the AI agents.

2. The Unknown Track, which was played on a multi-core processor under a parameterization known only after the competition deadline, since it was dependent on the result of a public lottery draw. The goal of the Base Track was to test agent capabilities in the standard Simplexity game. The Unknown Track evaluated the generalization power of the submitted solutions when applied to a most likely untested parameterization.

For each track, the submitted agents played against each other two times, so each had the opportunity of playing first. Agents were awarded 3 points per win, 1 point per draw and 0 points per loss, with the final standings for each track depending on the total number of points obtained per agent.

Classifications for the Base Track were updated daily during a five month period, up until the submission deadline, together with two larger test parameterizations. This allowed participants to have an idea of how their submission was faring, and update it accordingly.

The competition had a total of six submissions, four of which were from undergraduate students, both solo and in teams. Two of the submissions were from students of the author which fared well in the internal competition discussed in the previous subsection. Although the number of submissions was low, the fact that four of them were from undergrads, partially demonstrated that the competition was accessible

Building an AI agent for ColorShapeLinks is very simple, asking only the implementer to extend a class and implement one method. A basic Minimax algorithm with a simple heuristic can be implemented in less than 30 minutes. Educators can use it to demonstrate how to create a simple agent from scratch, during a class for example, and leave up to the students to find better, more efficient solutions. This can be done as an assignment, a competition, or both.

III. Related Work

Educational Context and Motivation ColorShapeLinks was originally developed as a Unity-only assignment for an AI for Games course unit1 at Lusofona University's Videogames BA.[1] This is an evenly interdisciplinary degree (Mateas & Whitehead, 2007), Hence, with the goal of increasing motivation among students. I participate to the competition to learned about AI and how I can develop my own Ai agent to win. by offering us the possibility of submitting our solutions to international events, and of comparing the solutions with the state of the art, our engagement and motivation are improved. Following this line of reasoning, ColorShapeLinks was proposed as a competition for the IEEE CoG 2020 conference. Since the first version of the framework was only suitable for basic classroom competitions, it was extended to support more advanced use cases. Additions included a console mode, for advanced debugging and analysis of agents, and a set of scripts for setting up automatically running tournaments.

The classical search approach in board game AI is the Minimax algorithm [4] [5]. This algorithm performs a depth-first search of the game tree down to a predefined search depth, bubbling up board evaluations obtained at maximum depth with a given heuristic function in a recursive fashion. Minimax evaluates game states at each depth from the AI's perspective, while assuming the adversary will also choose the best move for himself. As such, Minimax maximizes board evaluations when the AI is playing, minimizing them in the opponent's turn. Minimax has been optimized and improved through the years. Some of these optimizations, discussed for example in reference [4], are summarized in the following paragraphs.

Alpha-Beta pruning and Move ordering have been studied together in the past. The algorithm analyzes the most probable optimal movements first, improving the odds of pruning possibly poorer branches and making the system perform quicker and explore deeper. Iterative deepening is an optimization technique that works effectively when there is a time constraint on the optimization process itself. When it still has time to seek, it searches deeper and deeper. It effectively does a depth-first search in a breadth-first way, allowing the AI to spend as much time as possible without exceeding a predetermined time constraint.

A hash table and a collection of distinct integer numbers for each point on the board are utilized by the AI to transform a board state into a numerical identification that can be stored and retrieved quickly and efficiently. Along with the heuristic score for the corresponding board states, these data are kept in a hash table to avoid repeated evaluations of heuristic scores for those board states. The state of the art in board game AI can be considered a combination of Monte Carlo Tree Search (MCTS) and deep reinforcement learning [5] [4] [6]. According to the amount of wins, ties, and losses for each node, the MCTS algorithm simulates as many random play outs as feasible from any given board state while thinking time is available. Artificial neural networks are combined with reinforcement learning (e.g., QLearning) in deep reinforcement learning, allowing AI to learn the optimal moves in any given board state.

IV. Proposed Solution

A. *Implementation Details*

The SimpAI agent was developed in `C#` (.NET Standard 2.0 [7]), a requirement for the competition as the agents need to run both in the Unity game engine [1] and in the console. The implementation is divided into two different parts,which work together to form SimpAI:

- The minimax algorithm, used to search for promising future moves in the time it has available to think, discussed in Subsection IV-4
- 2. The heuristic one utilized to identify future worker process according to their strategic importance, leading the search algorithm to discover the optimum solution. The heuristic is presented in Subsection Subsection IV-5

In practice, Minimax combined with the heuristic function using a combination of several multi-threads (tasks) is something that we want to try in Subsection IV-B-1 in order to build our AI agent.

1) Execute an AI agent: :

The first step to implement an AI agent is to extend the AbstractThinker base class. This class has three overridable methods, but it is only mandatory to override one of them, as shown in Table bellow. There is also the non-overridable OnThinkingInfo() method, which can be invoked for producing "thinking" information, mainly for debugging purposes. In the Unity frontend this information is printed on Unity's console, while in the console frontend the information is forwarded to the registered thinker listeners (or views, from a MVC perspective). Classes extending AbstractThinker also inherit a number of useful read-only properties, namely board and match configuration properties (number of rows, number of columns, number of pieces in sequence to win a game, number of initial round pieces per player and number of initial square pieces per player) and the time limit for the AI to play. Concerning the board/match configuration properties, these are Fig.[3]

Method	Mandatory override?	Porpuse
Setup()	No	Setup the AI agent.
Think()	Yes	Selected the next move to perform.
ToString()	No	Return the AI agent's name.

TABLE I: Overridable Methods in the AbstractThinker class.

also available in the board object given as a parameter to the Think() method. However, the Setup() method can only access them via the inherited properties. the following subsections address the overriding of each of these three methods.

2) Overriding method:

- setup():
  If an AI agent needs to be configured before starting to play, the Setup() method is the place to do it. This method receives a single argument, a string, which can contain agent-specific parameters, such as maximum search depth, heuristic to use, and so on.[3] It is the agent's responsibility to parse this string. In the Unity frontend, the string is specified in the "Thinker params" field of the AIPlayer component. When using the console frontend, the string is passed via the –white/redparams option for simple matches, or after the agent's fully qualified name in the configuration file of a complete session. Besides the parameters string, the Setup() method also has access to board/match properties inherited from the base class. The same AI agent can represent both players in a single match, as well as more than one player in sessions/tournaments. Additionally, separate instances of the same AI agent can be configured with different parameters. In such a case it might be useful to also override the ToString() method for discriminating between the instances configured differently. This is an essential feature if ColorShapeLinks is running under a machine learning and/or optimization infrastructure. Note that concrete AI agents require a parameterless constructor in order to be found by the various frontends. Such constructor exists by default in `C#` classes if no other constructors are defined. However, it is not advisable to use a parameterless constructor to setup an AI agent, since the various board/match properties will not be initialized at that time. This is yet another good reason to perform all agent configuration tasks in the Setup() method [2]. In any case, concrete AI agents do not need to provide an implementation of this method if they are no parameterizable or if they do not require an initial configuration step.
- Think():
  The Think() method is where the AI actually does its job and is the only mandatory override when extending the AbstractThinker class. This method accepts the game board and a cancellation token, returning a FutureMove object. In other words, the Think() method accepts the game board, the AI decides the best move to perform, and returns it. The selected move will eventually be executed by the match engine. The Think() method is called in a separate thread. Therefore, it should only access local instance data.[3] The main thread may ask the AI to stop thinking, for example if the thinking time limit has expired. Thus, while thinking, the AI should frequently test if a cancellation request was made to the cancellation token. If so, it should return immediately with no move performed. The game board can be freely modified within the Think() method, since this is a copy and not the original game board being used in the main thread. More specifically, the agent can try moves with the DoMove() method, and cancel them with the UndoMove() method. The board keeps track of the move history, so the agent can perform any sequence of moves, and roll them back afterwards. For parallel implementations, the agent can create additional copies of the board, one per thread, so that threads can search independently of each other [2]. The CheckWinner() method of the game board is useful to determine if there is a winner. If there is one, the solution is placed in the method's optional parameter. For building heuristics, the game board's public read-only variable winCorridors will probably be useful. This variable is a collection containing all corridors (sequences of positions) where promising or winning piece sequences may exist. The AI agent will lose the match in the following situations:
  - Causes or throws an exception.
  - Takes too long to play.
  - Returns an invalid move.

- ToString():
  By default, the ToString() method removes the namespace from the agent's fully qualified name, as well as the "thinker", "aithinker" or "thinkerai" suffixes.[3] However, this method can be overridden in order to behave differently. One such case is when agents are parameterizable, and differentiating between specific parameterizations during matches and sessions becomes important [2].

3) Some included Agent: :
Three test agents are included with the framework, serving both as an example on how to implement an agent, as well as a baseline reference for testing other agents.[3] The sequential agent always plays in sequence, from the first to the last column and going back to the beginning, although skipping full columns. It will start by using pieces with its winning shape, and when these are over, it continues by playing pieces with the losing shape. Therefore, it is not a "real" AI agent. The random agent plays random valid moves, avoiding full columns and unavailable shapes. It can be parameterized with a seed to perform the same sequence of moves in subsequent matches (as long as the same valid moves are available from match to match). [2] The minimax agent uses a basic, unoptimized Minimax algorithm with a naive heuristic which privileges center board positions. It can be parameterized with a search depth, and, although simple, is able to surprise unseasoned human players—even at low search depths.

This section discusses the application of the heuristic function of a Minimax AI thinker. A minimax Algorithm for a two-player game is a 'recursive algorithm for selection of the next move.' As the search area otherwise would be too big, the basic versions of this algorithm try to take all potential motions by branching the game thread to maximum depth. The majority of board states that have been looked for by the algorithm are not final boards, even at the greatest depth. Therefore, to assess these non-final boards, we require a heuristic function. A heuristic is "an intelligent estimate, an instinctive assessment" that helps us judge the "goodness" of an administrative situation.

4) MinimaxAlgorithm: :
A minimax algorithm works by maximizing the heuristic score of all possible moves when it's the AI's turn to play, and minimizing it when it's the opponent's turn. Sample AI thinker using a basic Minimax algorithm with a naive heuristic which previledges center board positions as it's shown in fig 2. As such, a Minimax() function requires:

- The current board state.
- The color of the AI player.
- The color of who's playing in the current turn.
- The current depth.
- The maximum depth.

It will also need the CancellationToken, so it can check for cancellation requests from the main thread.

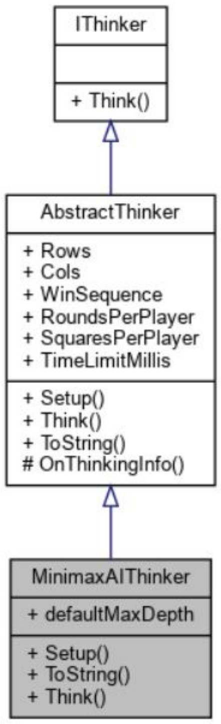


Fig. 2: Inheritance diagram for MiniMaxAIthinker. [3]

The infrastructure is all set. The following steps have to be implemented in the Minimax() function:

- If the cancellation token was activated, return immediately with a "no move" (score is irrelevant).
- Otherwise, if the board is in a finalstate, return the appropriate score (move is irrelevant since no moves can be made on a final board):
  - If the winner is the AI, return the highest possible score.
  - If the winner is the opponent, return the lowest possible score.
  - If the match ended in a draw, return a score of zero.
- Otherwise, if the maximum depth has been reached, return the score provided by the heuristic function (move is irrelevant, since the game tree will not be branched further below this depth, and as such, there's no move to chose from).
- Otherwise, for each possible move, invoke Minimax() recursively, selecting the best score and associated move (i.e., maximizing) if it's the AI's turn, or selecting the worst score and associated move (i.e., minimizing) if it's the opponent's turn.

5) Heuristic function: :
This is a fundamental part of the solution, and as such, only a very basic approach is discussed here. Intuitively, pieces near or at the center of the board potentially contribute to more

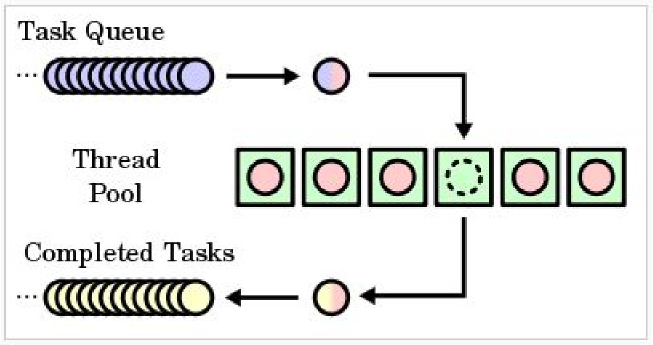


Fig. 3: A simple thread pool (green boxes) with waiting tasks (blues) and completed tasks (yellow)

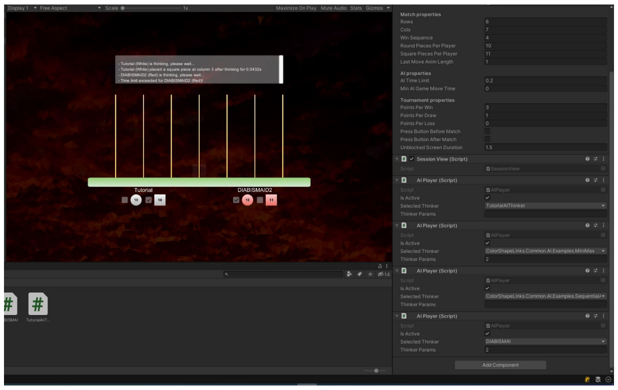
winning sequences than pieces near corners or edges. This is not a scientific claim, just a (possibly unfounded) guess.

First we search a distance between two point applying the following mathematical formula : $\sqrt{(x_1^2 + x_2^2) + (y_1^2 + y_2^2)}$, then we try to determine the center row. Afterwards, we maximize the points piece can be awarded when it's at the center. Next step we loop through the board searching for pieces till we get piece in current position and if the found piece is of our color, we increment the heuristic inversely to the distance from the center otherwise, decrements the heuristic value using the same criteria. Regarding the shape, if the piece match our shape, we increment the heuristic inversely to the distance from the center if it's not decrement the heuristic value using the same method. At the end we should return the final heuristic score for the given board.

6) Our AI agent: :

Our implementation required minimax function, heuristic function and in addition we used task which is .NET framework provides Threading. Tasks class to let you create tasks and run them asynchronously. A task is an object that represents some work that should be done. The task can tell you if the work is completed and if the operation returns a result, the task gives you the result. as it's shown in fig3. [7]

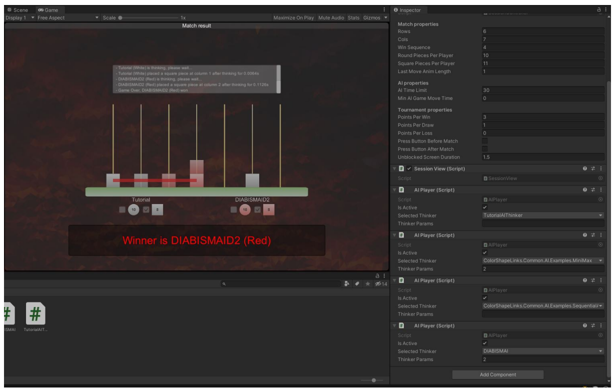
we set 7 task separated to each column, to search for the best move. Starting with searching solution index The search is conducted through a Minimax algorithm, with a number of optimizations, namely start index, finish index, board and depth, all of which were discussed in Section 3.1.1. The search was also parallelized, allowing other tasks to be evaluated once the task finish the other once start asynchronously by distributing the workload by the available CPU cores. In practice, this was achieved with the SelectedMoveForThreads method in `C#`, which internally optimizes the number of available worker threads according to the number of hardware threads. Since the Minimax family of search algorithms works recursively, and recursive algorithms are difficult to parallelize effectively [7], each possible move at the first depth level is processed iteratively in the main thread. When all the threads end their work the results are compared in the main thread



```
- Tutorial (White) is thinking, please wait...
- Tutorial (White) placed a square piece at column 3 after thinking for 0.0432s
- DIABISMAID2 (Red) is thinking, please wait...
- Time limit exceeded for DIABISMAID2 (Red)!
```

```
Match properties
Rows                        6
Cols                        7
Win Sequence                4
Round Pieces Per Player     10
Square Pieces Per Player    11
Last Move Anim Length       1
AI properties
AI Time Limit               0.2
Min AI Game Move Time       0
Tournament properties
Points Per Win              3
Points Per Draw             1
Points Per Loss             0
Press Button Before Match
Press Button After Match
Unblocked Screen Duration   1.5
Session View (Script)
Script                      SessionView
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            TutorialAIThinker
Thinker Params
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            ColorShapeLinks.Common.AI.Examples.MiniMax
Thinker Params              2
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            ColorShapeLinks.Common.AI.Examples.Sequential/
Thinker Params
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            DIABISMAI
Thinker Params              2
```

Fig. 4: win the game with max depth = 1 timelimit=0.2s



```
- Tutorial (White) is thinking, please wait...
- Tutorial (White) placed a square piece at column 1 after thinking for 0.0064s
- DIABISMAID2 (Red) is thinking, please wait...
- DIABISMAID2 (Red) placed a square piece at column 2 after thinking for 0.1126s
- Game Over, DIABISMAID2 (Red) won
```

Winner is DIABISMAID2 (Red)

```
Match properties
Rows                        6
Cols                        7
Win Sequence                4
Round Pieces Per Player     10
Square Pieces Per Player    11
Last Move Anim Length       1
AI properties
AI Time Limit               30
Min AI Game Move Time       0
Tournament properties
Points Per Win              3
Points Per Draw             1
Points Per Loss             0
Press Button Before Match
Press Button After Match
Unblocked Screen Duration   1.5
Session View (Script)
Script                      SessionView
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            TutorialAIThinker
Thinker Params
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            ColorShapeLinks.Common.AI.Examples.MiniMax
Thinker Params              2
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            ColorShapeLinks.Common.AI.Examples.Sequential/
Thinker Params
AI Player (Script)
Script                      AIPlayer
Is Active                   ✓
Selected Thinker            DIABISMAI
Thinker Params              2
```

Fig. 5: win the game with max depth = 2 timelimit=30s

for every given depth, and the branch with the best heuristic value decides the move that needs to be taken.

B. Experimental Results

in our experiment,we give our AI a maxdepth=1 with 200ms respecting all the required rules in the competition. Then we win all the matches. On the other hand, we noticed that the performence of our AI relative to the maximum depth and time limit.Once we increse the maxdepth we need to do the same thing regarding the time limit.

For example, we set maxdepth to 2 without changing the time limit, then the agent lose due to the long thinking time as it showns in fig4. therefore, we increment the time to 30s and we observe that our Ai Agent win as illustrated in fig5.

To explain this relationship between maxdepth and timelimit we need minimize the thread in our program test in order to get the optimum result even when we raise the number of depth or testing the AI agent in multiples CPU's Cores also which is not required in the competition.

V. Conclusion and Future Work

In this paper, we presented SimpAI, an AI agent created for the ColorShape- Links board game competition. The agent was implemented with an efficient Minimax-type search, and an heuristic function composed of 7 multi-threading task. SimpAI reached the second position in the base Track.

There are some elements that are likely to bring significant improvements to the agent's performance. For example, we minimize the task and change calling the Minimax() method from within SearchSolutionInIndexRange(), which will exponentially create threads.a wider initial set of partial heuristics could help refine what are indeed the most important features for winning Color-ShapeLinks matches.Furthermore, some partial heuristics employ ”magic” numbers to evaluate them. These might also be seen by the evolutionary algorithm as parameters to be tuned and further refine a winning approach.. In the optimization process a broader experimentation and parameters might perhaps lead to better successful combinations using the evolutionary operators.Finally, next step we will try next time to enhance our program using deep learning and/or reinforcement learning, could potentially produce stronger and harder to beat agents.

References

[1] N. Fachada, "Colorshapelinks: A board game ai competition for educators and students," *Computers and Education: Artificial Intelligence*, vol. 2, p. 100014, 2021.

[2] P. Fernandes, P. Inácio, H. Feliciano, and N. Fachada, "Simpai: Evolutionary heuristics for the colorshapelinks board game competition," *Videogame Sciences and arts, VJ*, 2020.

[3] N. Fachada, "Colorshapelinks ai an ai competition for the ieee conference on games 2021," 2021.

[4] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2018.

[5] G. N. Yannakakis and J. Togelius, *Artificial intelligence and games*. Springer, 2018.

[6] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *arXiv preprint arXiv:1811.12560*, 2018.

[7] "task-and-thread-in-c-sharp," 2021.