ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

First Practical Project: Hex - Board Game

Group 03

48280 André Jesus
48287 Nyckollas Brandão

Professor: Nuno Leite

Report written for Artificial Intelligence
BSc in Computer Science and Computer Engineering

April 2023

Abstract

This report outlines the design and implementation of a Prolog [1, 2] program for the board game, Hex [3].

Hex is a two player abstract strategy board game in which players attempt to connect opposite sides of a rhombus-shaped board made of hexagonal cells. Each player is assigned a pair of opposite sides of the board, which they must try to connect by alternately placing a stone of their color onto any empty hex. Once placed, the stones are never moved or removed. A player wins when they successfully connect their sides together through a chain of adjacent stones. Draws are impossible in Hex due to the topology of the game board. Despite the simplicity of its rules, the game has deep strategy and sharp tactics. It also has profound mathematical underpinnings.

The program allows for human-human and human-computer gameplay using the alpha-beta pruning algorithm [4], which efficiently implements the minimax principle [5].

Keywords: Prolog; Hex; backtracking; minimax principle; alpha-beta pruning.

Resumo

O presente relatório descreve o design e a implementação de um programa Prolog[1, 2] para o jogo de tabuleiro Hex [3].

Hex é um jogo de tabuleiro de estratégia abstrata para dois jogadores no qual os jogadores tentam conectar lados opostos de um tabuleiro em forma de losango, feito de células hexagonais. Cada jogador recebe um par de lados opostos do tabuleiro, que devem tentar conectar colocando alternadamente uma peça da sua cor em qualquer hexágono vazio. Uma vez colocadas, as peças nunca são movidas ou removidas. Um jogador vence quando consegue conectar os seus lados através de uma cadeia de peças adjacentes. Os empates são impossíveis no Hex, devido à topologia do tabuleiro de jogo. Apesar da simplicidade das suas regras, o jogo possui estratégia profunda e táticas precisas. Também tem fundamentos matemáticos profundos.

O programa permite a jogabilidade humano-humano e humano-computador, usando o algoritmo alfa-beta pruning [4], que implementa com eficiência o princípio minimax [5].

Palavras-chave: Prolog; Hex; backtracking; princípio minimax; algoritmo alpha-beta pruning.

Contents

Chapter 1

Introduction

Hex [3] is a two-player abstract strategy board game that has gained popularity in recent years due to its simple rules and complex gameplay. In this game, players take turns placing stones of their respective colors on a rhombus-shaped board made up of hexagonal cells. The objective of the game is to connect the opposite sides of the board assigned to each player by forming a continuous chain of stones of their color.

Hex's simplicity and mathematical underpinnings have made it an attractive subject for research in artificial intelligence and programming. In this report, we present the design and implementation of a Prolog [1, 2] program that allows for human-human and human-computer gameplay using the alpha-beta algorithm [4], an efficient implementation of the minimax principle [5].

The primary aim of this report is to demonstrate how to apply Constraint Logic Programming to implement a two-person, perfect-information game. The learning objectives of this work include mastering the syntax of Prolog programs, basic programming constructs such as atoms, numbers, compound terms, lists, operators, and arithmetic. Additionally, the work aims to provide an understanding of backtracking, typical built-in predicates, the minimax principle, and the alpha-beta algorithm.

1.1 Outline

The report is structured as follows. First, we introduce the game of Hex and its basic rules, objectives, and mathematical underpinnings. Next, we present our approach to solving the problem of implementing a Hex game in Prolog. We provide a detailed description of the main predicates and structures used in the program, and justify all decisions taken in the design process. In the third chapter, we present the experimental results of the project, including the results of the tests performed to evaluate the performance of the program. Finally, in the conclusions section, we summarize the main findings and contributions of the report. We reflect on the challenges and successes

encountered in the project, and discuss potential future work that could build on our approach to improve the Hex game's implementation.

Overall, the main objective of this report is to describe the solution we found for the project, explaining the structure and all decisions taken. We aim to provide a comprehensive understanding of how to implement a Hex game in Prolog, including the syntax and basic programming constructs, backtracking, typical built-in predicates, and the minimax principle. By presenting our approach and detailing the rationale behind each decision made, we hope to provide a valuable resource for other people studying the field of artificial intelligence and Prolog programming.

Chapter 2

Problem Analysis

In this section, we present the main problems and steps involved in the design and implementation of a Hex game in Prolog, justifying our decisions.

This section is structured as we follow the steps of the design process. First, we started by creating and displaying the board. Next, we implemented the gameplay in the player vs. player mode, without the win condition. Then, we implemented the win condition, and finally, we implemented the gameplay in the player vs. computer mode.

2.1 Creating the Board

To represent the board, we decided to use a list of lists, where each list represents a row of the board. Each cell in the board is represented by an integer, which can be either 0 (empty), 1 (player 1 piece - black), or 2 (player 2 piece - white).

The `create_board/2` predicate is used to create the board. It takes two arguments, the first one being the number of rows and columns of the board, and the second one being the board itself. The predicate uses the `create_board_r/3` and the `create_row/2` predicates to create the rows of the board.

2.2 Displaying the Board

To display the board, we implemented the `display_board/1` predicate. It takes the board as an argument and prints it to the console. The predicate uses the `display_column_letters/2` predicate to print the column letters at the top and bottom of the board, the `display_board_r/2` and the `display_row/1` predicates to print the rows of the board.

Other utility predicates were also implemented to help with the display of the board, such as `print_spaces/1`, `print_cell/1`, and `print_row/1`.

2.3 Gameplay

The game starts with the `play/0` predicate, calling the `get_game_mode/1` and the `get_board_size/1` predicates to get the game mode and the board size from the user. The `play/0` predicate then calls the `create_board/2` predicate to create the board, and the `play_game/2` predicate to start the game in the selected mode.

The `play_game_pvp_r` and the `play_game_pvc_r` predicates are used to play the game in the player vs player and player vs computer modes, respectively. They take the board and the current player as arguments.

The `make_move/2` predicate is used to make a move in the board, after the `validate_move/2` predicate has validated the move. Utility predicates such as `parse_move/3`, `replace/5`, `prompt_player/2`, `get_column_number/2`, `get_board_size/2` and `get_cell/4` were also implemented to help with the gameplay.

After each move, a check is made to see if the game has ended. This is done by calling the `check_win/2` predicate, which checks if the current player has won the game.

2.4 Winning the Game

The `check_win/2` predicate is used to check if the current player has won the game. It takes the board and the current player as arguments. Then, if the player is 1, it calls the `check_top_bottom/1` predicate, and if the player is 2, it calls the `check_left_right/1` predicate. These predicates check if the current player has connected the top and bottom sides of the board, or the left and right sides of the board, respectively.

The logic behind this is that, if the current player has connected the top and bottom sides of the board, then the current player has won the game. Similarly, if the current player has connected the left and right sides of the board, then the current player has won the game. The algorithm first calculates the starting nodes (the nodes in the top or left side of the board) and then, using the `dfs/4` predicate, it checks through a depth-first search if the starting nodes are connected to the ending nodes (the nodes in the bottom or right side of the board).

We decided to implement a DFS algorithm because it is a simple and efficient algorithm to traverse a graph in a depth-first manner. Using a DFS algorithm, we can check if the current player has won the game in a reasonable amount of time, even for large boards.

To do this, utility predicates were implemented, such as `get_start_nodes/3`, `start_node/2`, `end_node/2` and `get_adjacent_cells/4`.

2.5 Computer Player

The computer player is implemented using the minimax algorithm with alpha-beta pruning. The `play_game_pvc_r/2` predicate is used to play the game in the player vs computer mode. It takes the board and the current player as arguments, working in a similar way as the `play_game_pvp_r/2` predicate, but when it is the computer's turn, it calls the `get_computer_move/4` predicate to calculate the best move for the computer.

This predicate uses the `minimax/5` predicate to calculate the best move for the computer. The `minimax/5` predicate takes the board, the current player, the board size and the best move as arguments. It first calls the `max_value/9` predicate, which in turn calls `min_value/9` predicate, and they call each other, recursively, to calculate the best move for the computer. For the loops of min in max, `max_value_loop/11` and `min_value_loop/11` exist.

This predicates implement the alpha-beta pruning algorithm, which is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. The alpha-beta pruning algorithm is an adversarial search algorithm used commonly for machine playing of two-player games, such as Hex. Our implementation is highly based on the traditional approach provided by the pseudo-code for the alphabeta algorithm.

The score provided by the leaves of the generated tree is based on whether or not the last move won the game, using the `check_win/2` predicate, for min or for max. In case max wins, a positive score based on the Depth is given (the higher the depth number the better, meaning game ended earlier). In case min wins, a negative score based on the Depth is given (the lower the depth number the better, meaning game ended earlier). If depth reaches 0, in either case, `evaluate_board/4` is used to determine a score based on the current state of the board, even if the winning condition does not apply. This was not implemented and as such, the score 0 is given (using this has the side effect of, if every single leaf ends in depth, the first possible move is chosen).

The `get_all_possible_moves/3` predicate is used to get all the possible moves for the current player. It takes the board and the board size as arguments, and returns a list of all the possible moves for the current player.

The `minimax_depth_by_board_size/2` predicate is used to obtain a depth based on the board size. This is useful, since the higher the board size, the more iterations the alphabeta algorithm will have to make. Therefore, the depth does not control the number of iterations consistently among all possible board sizes and is made custom based on the board size on our implementation. A future improvement to this could be obtaining the depth based on the current amount of empty cells on the board, evolving

as the game is played.

Chapter 3

Experimental Evaluation

In this chapter, we present the results of the experiments we conducted to evaluate the performance of our implementation of the Hex game. We also present the results of the experiments we conducted to evaluate the performance of the minimax algorithm with alpha-beta pruning, and the performance of the minimax algorithm without alpha-beta pruning.

Initially, we tested the performance of the game in the player vs player mode, using different board sizes. We tested the game using a board size from 3 to 11 (the maximum board size allowed by the game). We concluded that the game is very fast, even for large boards. The game is able to run in less than 0.0001 seconds for a board size of 11, because all predicates are implemented in a very efficient way.

3.1 Player vs Player Mode

The most time-consuming predicate in a PvP game is the `check_win/2` predicate, which is used to check if the current player has won the game. This predicate is called after each move, and it is very important to make sure that it is implemented in an efficient way. We implemented the `check_win/2` predicate using a DFS algorithm, which is a simple and efficient algorithm to traverse a graph in a depth-first manner. Using a DFS algorithm, we can check if the current player has won the game in a reasonable amount of time, even for large boards.

3.2 Player vs Computer Mode

After that we tested the performance of the game in the player vs computer mode, using a implementation of the minimax algorithm without alpha-beta pruning. We concluded that the game is reasonably fast for small boards, like a 2x2 or a 3x3 board, but it is very slow for large boards, when using a big depth number. This is because the minimax algorithm without alpha-beta pruning is a very slow algorithm, and it is

not able to calculate the best move for the computer in a reasonable amount of time, even for small boards.

Finally, we tested the performance of the game in the player vs computer mode, using a implementation of the minimax algorithm with alpha-beta pruning. We concluded that even with the pruning, when not defining an adequate depth, the game can be slow for large boards, because for each iteration of the algorithm, we call the win validation multiple times, which is a very time-consuming operation. In addition to that, the number of iterations can spiral out of control, because it evolves as a factorial based on the number of empty cells.

Having defined a custom depth for each board size, we had a bigger control over the performance, but we had to compromise on the intelligence of the computer: in the first few moves, depending on the board size, the computer may only play the first possible move, as all give the score of 0 (depth reached 0). However, as more moves are played, earlier winning paths may appear before depth reaches 0. This means the computer will have these in mind and appear to make smart decisions. As said before, obtaining the depth based on the current amount of empty cells on the board instead may be a better option.

We believe that the algorithm can be improved by using other factors to evaluate the best move for the computer, such as if the move is close to the center of the board, or if the move is close to the computer pieces.

Chapter 4

Final Remarks

In summary, the main objective of this project was successfully achieved. We were able to implement a Hex game in Prolog, learning the syntax and basic programming constructs, backtracking, typical built-in predicates, and the minimax principle. By presenting our approach and detailing the rationale behind each decision made, we hope to provide a valuable resource for other people studying the field of artificial intelligence and Prolog programming.

The main challenges we faced were related to the implementation of the computer player, where we had to implement the alpha-beta pruning. Other challenges included the implementation of the `check_win/2` predicate, where we had to implement a depth-first search to check if the current player had connected the top and bottom sides of the board, or the left and right sides of the board.

The project was challenging, but also very rewarding. We learned a lot about Prolog and Constraint Logic Programming. We also had a lot of fun implementing the game, and we are very happy with the final result.

Bibliography

[1] I. Bratko, *Prolog programming for artificial intelligence*. Addison Wesley, 2012.

[2] "Swi-prolog documentation," www.swi-prolog.org. [Online]. Available: https://www.swi-prolog.org/pldoc/doc_for?object=root

[3] W. Contributors, "Hex (board game)," Wikipedia, 01 2019. [Online]. Available: https://en.wikipedia.org/wiki/Hex_(board_game)

[4] ——, "Alpha–beta pruning," Wikipedia, 03 2019. [Online]. Available: https://en.wikipedia.org/wiki/Alpha\%E2\%80\%93beta_pruning

[5] ——, "Minimax," Wikipedia, 09 2019. [Online]. Available: https://en.wikipedia.org/wiki/Minimax