intelligent connect 6

dr/amr s.ghonim

*Faculty of Computers & Artificial Intelligence | Helwan University*

|  |  |  |  |
| --- | --- | --- | --- |
| department | year | id | STUDENT NAME |
| Is | 3 | 20210137 | اروى شريف محمد احمد |
| Is | 3 | 20210151 | اسراء سعد حافظ محمد |
| Is | 3 | 20210153 | اسراء علي سيد محمود |
| Is | 3 | 20210188 | اندرو ايمن انسي عبدالملاك |
| Is | 3 | 20210142 | اسامه محمود عز الدين محمد |
| Is | 3 | 20210104 | احمد محمد عبدالمعز عبدالستار |
| Ai | 3 | 20210133 | ادم محمد عطيه عبدالغني |

Project overview

INTRODUCTION

Artificial Intelligence (AI) has a rich history in game research, with one of the pioneering areas being computer chess. The potential perceived in computers led researchers to explore tasks requiring human-like intelligence. This exploration paved the way for advancements in game tree search techniques that have broader applications in computer science and AI beyond the realm of chess (Allis, 1988).

Extensive research has been dedicated to the field of computer chess, revealing insights specific to chess and others with broader implications. The difficulty in developing programs that mimic human chess playing strategies became evident, prompting a shift towards full-width or brute-force searching on high-speed hardware, as suggested by Buro in 2002.

The context of this research involves a two-player zero-sum game with perfect information, where the outcome reflects perfect play from both sides. The game's value is determined by recursively exploring all possible continuations from each state until a known outcome state is reached, utilizing the minimax rule to propagate values to the initial state. This exploration forms a game tree, with the root as the initial state and leaf nodes as final states. Game tree search aims to mitigate errors in the evaluation function by growing a

full-width tree as deep as time allows. The alpha-beta algorithm efficiently implements this approach (Baum & Smith, 1997).

In this project, the focus is on the traditional application of AI, simulating gameplay for the Connect Six game on a 19x19 board. Connect Six, a larger and more intricate variant of Connect Four, is a zero-sum adversarial game, where advantages for one player translate into disadvantages for the opponent. The project emphasizes adversarial search in the context of AI, exploring challenges and solutions specific to planning ahead in a game where the player competes against a sophisticated AI adversary.

Our project revolves around the development of an artificial player for Connect Six, a strategic board game that engages two players in a challenge of wit and tactics. it is similar to connect4 game and gomokou game. Connect Six introduces an expanded grid, featuring 19rows and 19 columns. Participants select a colour and alternate turns, player choose a vertex to put his colour in , securing the lowest available slot within the chosen column. Victory is achieved by forming a line of six discs in a row, whether horizontally, vertically, or diagonally.

Connect Six poses an intriguing challenge as it extends the dimensions of traditional Connect Four, and similar to its predecessor, Connect Six is a solved game. The inherent complexity makes it an ideal candidate for implementing artificial intelligence strategies. Our project will leverage the Alpha-Beta pruning algorithm, integrating multiple heuristic functions to enhance decision-making. The objective is to evaluate and identify the most effective heuristic function through experimentation and analysis.

For the development, we have chosen Python 3.11 as our programming language, ensuring flexibility and ease of implementation. Additionally, we will employ essential libraries such as NumPy and Pygame to streamline the coding process and create an interactive and visually appealing gaming experience. The incorporation of sophisticated algorithms and heuristic evaluation will empower our artificial player to make informed and strategic decisions, providing an engaging challenge for human opponents

Main functionalities

* **GAME SETUP**

There's no setup needed for the game. It's played on a square board made up of orthogonal lines, with each intersection capable of holding one stone. The game board can be any finite size from 7x7 up (integers only),and extremely large or infinite boards are of little practical use. 19x19 Go boards might be the most convenient.

* **PLAYERS MOVE**

Two players, Black and White, alternately place two stones of their own color, black and white respectively, on empty intersections of the board. Black (the first player) places one stone only for the first move. Subsequently, White and Black take turns, placing two stones on two different unoccupied spaces each turn (IN OUR CASE ONE PLAYER WOULD BE THE AGENT )

* **Winning Condition**

The player who is the first to get six stones in a row (horizontally, vertically, or diagonally) wins.

* Features of connect 6

1. Choosing colour of your stone
2. Play with a friend
3. Play with AI
4. Choose the difficulty of the game with the AI

A diagram of a game

Description automatically generated

**Play With AI description**

|  |  |
| --- | --- |
| **Identifier & Name** | player *play with ai* |
| **Initiator** | *player* |
| **Goal** | Player is able to play connect6 with AI |
| **Precondition** | None |
| **Postcondition** | player has to pick his chosen color & difficulty |
| **Main Success Scenario** | 1. player presses play button 2. player picks his desired color 3. player picks the difficulty of AI 4. player starts playing the game 5. After finishing game player returns to Main Menu |
| **Extensions** | 2.a player presses back  2.a.1 User goes back to main menu  3.a player presses back  3.a.1 User goes back to Choose Color screen |

**Play With Friend description**

|  |  |
| --- | --- |
| **Identifier & Name** | *Player play with friend* |
| **Initiator** | *Player* |
| **Goal** | *Player 1 is able to play connect-4 with*  *Player 2* |
| **Precondition** | None |
| **Postcondition** | Player 1 has to pick his chosen color |
| **Main Success Scenario** | 1. *Player* presses play with friend button 2. Player 1 picks his desired color 3. Player 1 & Player 2 start playing the game 4. After finishing game they return to Main Menu |
| **Extensions** | 2.a *Player* presses back  1.a.1 User goes back to main menu |

**Choose Color description**

|  |  |
| --- | --- |
| **Identifier & Name** | *player choose color* |
| **Initiator** | *player* |
| **Goal** | *player chooses red or blue as his disc color* |
| **Precondition** | player chose to play game |
| **Postcondition** | player has to pick difficulty of AI or None if local |
| **Main Success Scenario** | player |
| **Extensions** | 1.a player presses back  1.a.1 player goes back to main menu |

**Choose Difficulty description**

|  |  |
| --- | --- |
| **Identifier & Name** | *player choose difficulty* |
| **Initiator** | *User* |
| **Goal** | *player chooses easy, medium, or hard as AI difficulty* |
| **Precondition** | player chose to play game and User chose his color |
| **Postcondition** | player plays game |
| **Main Success Scenario** | 1- player picks either easy, medium, or hard |
| **Extensions** | 1.a player presses back  1.a.1 player goes back to choose color screen |

**Quit Game description**

|  |  |
| --- | --- |
| **Identifier & Name** | *player quit game* |
| **Initiator** | *player* |
| **Goal** | *Game closes successfully* |
| **Precondition** | None |
| **Postcondition** | Program is terminated successfully |
| **Main Success Scenario** | 1. player presses quit 2. Game window is closed and terminated |
| **Extensions** | None |

Similar application

* CONNECT 6 ONLINE(web) <https://sdin.jp/en/browser/board/connect6/>
* CONNECT 6 (mobile ) <https://play.google.com/store/apps/details?id=jp.sakeapps.rokumoku&hl=en_US>
* GOMOKU ( same but connect 5 in row) <https://papergames.io/en/r/ua-o73jdRc>
* TIC TAC TOE (3 in row )

https://playtictactoe.org/

Literature review

Abstract

The field of artificial intelligence has come a long way in the last 50 years, and studies of its methods soon expanded to a field in which they are of great practical value—computer games. The concept of intelligent agents provides a much needed theoretical background for the comparison of various different approaches to intelligent, rational behaviour of computer-controlled characters in games. By combining rationality with certain limitations to the capabilities of our agents, we can achieve behaviour resembling that of a human player, which is also desirable in games. The goal of this article is to introduce various types of agents that are used in games, show how to implement meaningful, reasonable limitations to agent capabilities into the game world, and provide a freely available, open-source application for the comparison of such agents. Additionally, in this article we show that even the simplest agents can succeed in their tasks in certain task environments, whereas more difficult task environments often require a more sophisticated agent architecture. Our application consists of two task environments with nine agents in total but could easily be extended with additional task environments and agent implementations. In the end, we find the addition of goals into the agent architecture has the biggest impact on the agent's behaviour and performance, whereas the state-based approach helps keep our implementation simple and compact.

## Introduction

The field of artificial intelligence is very broad. Its methods are used in economics, mathematical problems and proofs, medical diagnosis, chess, etc. In this chapter we focus on its use in computer games, where its main use is control of nonplayer characters (NPCs). Our goal is to introduce and compare various types of agents used in the game world. For the purpose of comparison of agents we developed two scenarios, and the entire implementation is available as an open-source tool. The application and this accompanying chapter are intended to provide an educational review, comparison and evaluation, similar to [1].

We first define the task environment (Section 2) and present an overview of the different types of intelligent agents in games, based primarily on the categorization popularized by Russel and Norvig [2] (Section 3). Then we define the test scenarios (Section 4) and implement (Section 5) and compare (Section 6) various intelligent agents that will have to perform the tasks specified by the scenarios.

We use the freely available development platform Unity3D [3] to first create the desired environments and then implement the agents. Our application serves as a foundation for future implementation of new agents, enabling their comparison in already given scenarios, or in newly designed ones. The source code is freely available under the GPL [4] on the GitHub repository [5]. It may be used freely for any future research, educational purposes, or other use, and the repository also gives even the users without programming knowledge the ability to suggest improvements and additional features. The program has been tested in Windows, but, due to Unity's platform independence, should be transferable without major problems to other operating systems.

An agent is a program that acts autonomously in a given environment (which can be virtual or physical). It receives information about the state of the world around it in the form of precepts through sensors, which can again be either virtual or physical. It then responds to the information by choosing an appropriate action and executing it via its actuators. This process is illustrated in Fig. 1. Formally, the choice of action can be defined as a mathematical function, called the agent function, which maps every possible percept sequence to an action.

This definition of an agent is only one of many. Franklin and Graesser [6] demonstrate the plethora of different formal definitions of a software agent and seek to unite their essence. Nwana [7] delves deeper into the subject. Likewise, there are various ways to define intelligence; a more in-depth discussion on the topic from the AI community can be found in [2, 8, 9].

Throughout this article, we adopt Russel and Norvig's definition. Informally, intelligence (also known as rationality) refers to the agent's ability to always choose the action it considers to be the best in the current situation. For a formal definition we first introduce the performance measure, which rates the agent's success in performing the task it was given in its task environment. Given its dependence on the specific task, it is generally best if we define the performance measure function ourselves when we define the tasks (thereby defining what we consider to be good, desirable behavior), rather than using a predefined fixed one (see [10]). By convention, the higher the agent's performance measure, the better it performed its task.

Formally, then, an intelligent agent always chooses the action, which will maximize its expected performance measure, given its current percept sequence and its built-in knowledge of the task environment. Note that the performance measure need not be explicitly known to the agent for it to be able to make intelligent decisions, and in fact most of the agents we mention have no knowledge of it. They are, however, still capable of making fairly intelligent (or in some cases even perfectly intelligent) decisions in various task environments.

Note that, at least following this definition, our measure of intelligence depends on the given task environment. Therefore, many agent types may indeed achieve intelligence in certain (simpler) task environments but not in more complex ones.

The game world is a virtual environment with rules and laws, affecting the characters and objects in it. These rules need not apply equally to every character, though at least the laws of physics normally do. The characters can be player-controlled (player characters) or computer-controlled (NPCs). Note that the characters are entities with visible bodies and require only that their behavior appear rational and autonomous to the human player. Thus, the simpler ones can often be controlled by programs that do not in fact exhibit true intelligence as we defined it. The intelligent agent approach is merely one of the ways to control the characters (for further discussion on this subject, see [11]). It is, however, a reliable approach, since it guarantees that the characters will indeed appear both intelligent and autonomous, as true intelligence and autonomy are the very requirements we set in our definition of an intelligent agent.

Besides intelligence we usually require one more property of our agents in the game world: fairness. The computer used to control characters will generally at all times have complete knowledge of the current world state. Ideally, however, we would like it to control every character using only information that particular character can currently perceive, according to the rules we have set in our game world. Therefore, we restrict the information available to each of the agents, and in doing so, we in fact programmatically implement the agent's sensors (virtual sensors). Examples of the agent's actions, in turn, are moving around, communicating with other agents, assisting or attacking (hindering) them, etc.

With reasonable restrictions our agents then cannot see through walls, cannot hear sounds miles away, and ideally do not have direct access to the precepts of other agents.

#### The Minimax Algorithm

The problem is to design or propose a solution for this connect-4 game. This game can be solved very easily for two players by using matrices or arrays. However, a solution for one player against computer bot (AI) is proposed.

A Mini-Max algorithm is used to solve this game. The user is playing against a computer that uses the minimax algorithm to generate the game state. Mini-Max falls into the category of backtracking algorithms. This algorithm has many uses and is used in decisionmaking and game theory to find the optimal move for a player when the opponent is also playing optimally.

It is commonly used in two-player games, such as tic-tac-toe and chess, where players take turns making moves. there are two players in mini-max which are called maximizer and minimizer.

Maximizer aim to get the highest possible score, while Minimizer do the exact opposite and try to get the lowest possible score. Each board state is assigned a value. In certain states, the board score tends to be positive when Maximizer dominates. If Minimizer dominates in this board state, it tends to be a negative value.

#### The Alpha-Beta Algorithm

A proposed solution to this is to try and optimize the minimax algorithm, and that can be done by applying the Alpha-Beta algorithm (Nasa, Didwania, Maji, & Kumar, 2018). Alpha-beta is the most common way to search game trees in adversarial board games such as chess, checkers, and Othello. It's much more efficient than a simple brute-force minimax search because it allows you to prune large parts of the game tree while still ensuring the correct game tree values. However, the number of nodes visited by the algorithm increases exponentially as the search depth increases.

This obviously limits the scope of the search, as the game program must satisfy external time constraints. often you have only a few minutes to make a decision. In general, the more forward-looking the program, the better the quality of the play.

Alpha-beta pruning is not a new algorithm. It is basically an advancement or technique of the minimax algorithm that reduces the time complexity of the minimax algorithm by half and speeds up the AI bot's decision time for one move.

Applied algorithms

* Implementing minimax algorithms
* Pseudocode

01 Procedure Minimax(board, depth, alpha, beta, maximizingPlayer)

02 Set valid\_locations to the result of get\_valid\_locations(board)

03 Set is\_terminal to the result of is\_terminal\_node(board)

04 If depth equals 0 or is\_terminal is true then

05 If is\_terminal is true then

06 If winning\_move(board, AI\_PIECE) is true then

07 Return (None, 100000000000000)

08 Else if winning\_move(board, PLAYER\_PIECE) is true then

09 Return (None, -10000000000000)

10 Else

11 Return (None, 0)

12 End If

13 Else

14 Return (None, score\_position(board, AI\_PIECE))

15 End If

16 End If

17 If maximizingPlayer is true then

18 Set value to -∞

19 Set column to a random choice from valid\_locations

20 For each col in valid\_locations do

21 Set row to the result of get\_next\_open\_row(board, col)

22 Set b\_copy to a copy of board

23 Call drop\_piece(b\_copy, row, col, AI\_PIECE)

24 Set new\_score to the second element of the result of

25 Minimax(b\_copy, depth - 1, alpha, beta, false)

26 If new\_score is greater than value then

27 Set value to new\_score

28 Set column to col

29 End If

30 Set alpha to the maximum of alpha and value

31 If alpha is greater than or equal to beta then

32 Break

33 End If

34 End For

35 Return column, value

36 Else

37 Set value to ∞

38 Set column to a random choice from valid\_locations

39 For each col in valid\_locations do

40 Set row to the result of get\_next\_open\_row(board, col)

41 Set b\_copy to a copy of board

42 Call drop\_piece(b\_copy, row, col, PLAYER\_PIECE)

43 Set new\_score to the second element of the result of Minimax(b\_copy, depth - 1, alpha, beta, true)

44 If new\_score is less than value then

45 Set value to new\_score

46 Set column to col

47 End If

48 Set beta to the minimum of beta and value

49 If alpha is greater than or equal to beta then

50 Break

51 End If

52 End For

53 Return column, value

54 End If

55 End Procedure

* Flowchart

A diagram of a flowchart

Description automatically generated

Development platform

• USED TOOLS

-visual studio code

• USED LANGUAGES

- Python

• USED LIBRARIES

- numpy

- random

- pygame

- sys

- math

- tkinter

-link to the shared folder -

<https://drive.google.com/drive/folders/1tG4xwVAyC8-V6WPzUV1Km6PovwTVVGJZ?usp=drive_link>