How do Artificial Intelligence (AI) agents strategize in games with imperfect information?

Abby Smith

u2718717

Abstract

I made a top-down strategy game featuring two AI opponents using different AI algorithms, with one opponent using Minimax and the other using Hierarchical Portfolio Search (HPS). The AI's win rates were recorded. The AI agents won against most new human players and provided a good challenge for experienced players. From this project I learned game engine programming, AI programming for advanced strategy games, and researched more advanced AI algorithms that can be used very effectively in simpler strategy games.

Project Summary

The goal of this project was to research AI programming for top-down strategy games with imperfect information. I had twelve weeks to complete this project over the course of the second term. I decided to make a top-down strategy game using C++ and the Raylib graphics library, written in Microsoft Visual Studio. This no-engine approach allowed me to cleanly construct the game out of my own code with no extraneous features. I tasked myself with completing the strategy game, implementing multiple AI algorithms, and

collecting playtest data about their effectiveness. During development, the scope had to decrease slightly, as I was too ambitious with the number of expected AI algorithms and with the ease of developing the game. After development, I held a playtest in which human participants played the game against AI opponents and collected data about their win rates. This allowed me to measure the success of the project, as well as which AI algorithms were more effective in the game. From this project, I have gained key AI programming skills, as well as general C++ and object-oriented programming skills.

Figure 1. Tutorial Portion of Strategy Game

Approach and Objectives

The major goals of this project were to gain an understanding of strategy game AI programming, its implementation, and the effectiveness of various decision-making algorithms. The three project milestones were the completed strategy game, AI implementation, and the playtest.

The strategy game was made in C++ using the Raylib graphics library for a 2D top-down view. Building the game in a low-level environment allowed me to limit the scope of the game to its own specific features and keep the codebase very simple. To effectively test the AI the game followed these design philosophies:

- The game features imperfect information, meaning that parts of the game are hidden from one or more of the players (Doyen and Raskin, 2011). In this case, players can't see tiles that are too far away from their units. This prevents the AI from being able to accurately predict the end result of the game as it cannot see what the human player is doing.
- The game has no "output randomness", defined as random results chosen after a decision is made, like an attack missing (Brown, 2020). This allows the AI to be sure of the results of individual moves.
- The game features "input randomness", defined as random results chosen before any decisions are made (Brown, 2020). Each game takes place on a random map, with a different layout of paths, walls, and castles. This forces the AI to utilize different strategies in each game.
- The game is as symmetrical as possible. Both players start on opposite sides of a symmetrical map with the same starting units. The only difference is that one player goes first, and the other receives some additional resources to improve fairness.
- The AI is unable to cheat in any way. Many games make their AI challenging by giving it an advantage over human players, which is not my goal.

I researched many algorithms for the AI programming portion of this project, including Monte-Carlo Tree Search, Hierarchical Portfolio Search, Dynamic Scripting, and Minimax. All of these techniques use the current game state (the positions of each unit, how many resources each player has, etc.) to determine a move, that is, a sequence of

game actions that makes up the entire turn (Sturtevant, 2015). Of these techniques, I implemented Minimax and Hierarchical Portfolio Search. These are both algorithms that run a search over their game actions to determine the best move. Minimax iterates over each unit, picking the best action from that unit's list of actions. This is a broad search that looks at every potential action but does not consider any actions in combination with each other (Sturtevant, 2015). HPS uses a portfolio of partial players to limit the number of actions it considers; these players each return an action for each unit. Then, the algorithm searches each combination of these actions for the best move. This is a deeper search that can find moves that work well in combination with each other but considers less actions overall (Churchill and Buro, 2017). While I aimed to produce three AI players, only two were able to be finished due to some technical difficulties (see Technical Challenges).

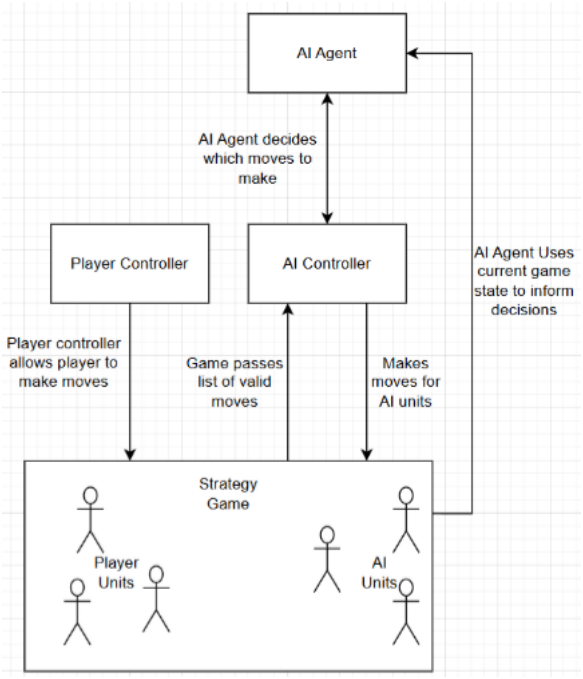

Figure 2. Implementation of AI agents within the strategy game

In the final milestone, I evaluated the efficacy of the AI agents. I performed a study to gather quantitative data about their performance, consisting of two stages: Tutorial and Data Collection. In the tutorial stage, players were instructed on how to play the game and played through an example match against a simple AI. This ensured that they had an understanding of the strategy game. Then, in the Data Collection stage, participants played a game against each AI agent, and their wins and losses were recorded, along with some additional survey data.

I sourced participants from the games course as they would already be familiar with strategy games, improving the effectiveness of the tutorial stage. A total of six players tested the game. Some of these subjects may have more or less familiarity with the game and the game genre than others, but this was not a significant problem as the skill level of the AI was being tested, not the skill level of the players.

The data shows that Minimax performed much better than HPS overall, with a 100% win rate over HPS' 67% win rate. This is because HPS performs a much narrower search and often makes slightly worse versions of moves Minimax would make. For example, HPS has an "Aggressive" partial player that suggests attacking the nearest target. In some cases, there are multiple tiles that targets could be attacked from, and the aggressive player simply picks one for the algorithm to consider. One play tester said, "The AI in the second game [HPS] decided not to create many units to attack with". This happened because its "Unit Purchasing" player chooses only the most expensive option, so it missed the opportunity to produce many smaller units. Meanwhile, Minimax considers every

possible spot to move and attack a target from, then picks the best option out of those. This means Minimax is much more likely to find the best position.

Time Management and Scope

The scope of this project was to develop a full top-down strategy game made in C++ with Raylib, including three AI players that would use different decision-making algorithms to select their moves. I had twelve weeks to finish the project, and I expected to be done with the strategy game and AI portions by week 8, so that I had time to conduct my playtest and finish the written reflection. In preparation for the project, I had already developed a prototype in Godot (see Figure 3), so I had a good understanding of what needed to be brought over into the C++ game. What I had not anticipated was the amount of time the basic game architecture, like creating, processing, and drawing objects, would take to develop. Ultimately, I wound up finishing the game closer to week 10, and had a lot less time for playtesting than I anticipated. I also had to cut back on some AI features due to technical difficulties, which will be discussed later in this paper. I was able to reach a very suitable minimum viable product in that I had a well-polished strategy game with more than one AI opponent.

Figure 3. Godot Prototype

Technical Challenges

The first technical challenge I encountered was that the game took longer to develop than anticipated. To support the main strategy game, I had to make a wide variety of basic game engine features. These complex systems resulted in much longer being spent on the game project. Part of the problem is that I overscoped; making a simpler game with fewer features would have taken much less time and still could have demonstrated AI programming skills. The other issue is that I chose to work with no engine, so I had to build up my own basic game architecture in order to build the rest off of it. If I had started from Godot or Unreal Engine, I could have saved a lot of time. However, I also learned a lot from programming this game from the ground up, so developing it this way had benefits too.

The second issue was the AI programming side of the project. Because the game was more complicated than it needed to be, getting the AI working was also a difficult task. AI algorithms are much easier to implement in simpler games, and the imperfect

information portion of my game provided additional challenges, as most algorithms are designed with perfect information in mind. Once I had a working Minimax AI, the problem became optimization. During the game, players expand their control over the map, gaining access to more castles and more units to command. This increase in options caused the AI to slow down, taking exponentially longer to complete its turns. To remedy this, I had to make multiple optimizations. I reduced the number of A* pathfinding calls the AI made on its turn, made the maps smaller to simplify the decision space, and moved texture loading to a singleton class that loads every texture at the start of the program, rather than loading textures when new units are created. These optimizations brought the turn time down greatly and made the game much more playable. I wanted to implement more advanced algorithms, however, currently, HPS takes a few seconds to calculate its turns even with these optimizations. Any deeper-layered search wouldn't run efficiently enough to be implemented.

In the future, I would like to make a simpler game that can more effectively support experimenting with advanced AI algorithms like Monte-Carlo Tree Search. This type of game is too complicated for working with more complicated AI techniques, which is why across the industry, most developers use simpler solutions. A game where players only move one piece per turn, like in chess, that also has perfect information, would be a much better environment for further research on this subject.

Professional Development and Career Relevance

Through this project, I have developed my AI programming skills and C++ programming skills, and have created a strategy game that makes a great portfolio piece. The AI skills are shown very clearly. I have learned a lot about integrating these algorithms into a strategy game, as well as the difference between broad shallow searches and narrow deep searches. This knowledge will improve my ability to implement systems like this in the future greatly. My C++ programming has also advanced a lot from this project. Making the game in pure C++ required me to create essentially an entire game engine, showing my advanced object-oriented programming skills. Finally, the strategy game makes a great portfolio piece. While the graphics are simple, it looks well put together. It doesn't have any major bugs or errors, and it is a great technical achievement. I plan to post multiple devlogs about this project on my website, going in depth about my strategy game AI learnings, as well as my game engine architecture and the design of my strategy game. I will also be sharing this project with a connection I made at EGX, who was very interested in seeing the end result.

Conclusion

In making the strategy game, I developed a very effective game engine in C++. Its features are heavily inspired by Godot, but the code is all my own, allowing for the development of a very lightweight game with no extraneous features. It supports a tree of objects with tick, input, and draw functions. These objects are used to create menus of buttons and the tile

map containing player units. It could even support a real-time game, and much more if I added a physics system. Going forward, I would love to make more games with this system.

The AI for the strategy game was very difficult to implement for such a complicated game. It needed to be able to understand complex game states with multiple moving pieces and hidden information, while having the right priorities in order to make the most effective moves. I learned that when the decision space is so large, a shallow, broad search like Minimax provides is more effective. This integration provided key insight into the AI for top-down strategy games and would be a great help if I were to make or work on a full-fledged game in this genre.

Finally, I studied more advanced AI algorithms that I wasn't able to use, like Monte-Carlo tree search and dynamic scripting. These would be much easier to implement in a simpler game with a smaller decision space. I aim to make a game that is more "chess-like" where players take only one action per turn. This would be a great future project to show off and advance my understanding in the field.

Appendices

“Castle Conquest” Rules:

Each player begins the game with a Knight unit and control of a Castle. Each turn, players may move each of their units by their movement speed, units can then attack enemy units within their attack range. Units can only see a certain distance; seen tiles are illuminated and the unit’s controller can see any units on those tiles. Enemy units that are too far away cannot be seen by a player, however castles can always be seen, providing some insight into the position of the enemy. Players also gain gold each turn, 100 per castle. This gold can be spent on purchasing new units at any castle. Units cannot move or attack on the turn they are purchased. Additionally, the player going second begins with 300 extra gold to improve fairness. There are three types of units, each with different prices and stats. A player wins when they gain control of all castles on the map. The full project is accessible to be played in the submission folder.

Playtest Data:

Trial Number	HPS Record	Minimax Record	HPS Score	Minimax Score
1	1	1	5	5
2	1	1	4	5
3	0	1	2	4
4	1	1	5	3
5	1	1	3	4
6	0	1	3	4
Average	0.666666667	1	3.666666667	4.166666667

This table contains the data from the strategy game playtest. “HPS Record” and “Minimax Record” refer to the wins and losses of the AI agents. A 1 represents a win while a 0 represents a loss. The “HPS Score” and “Minimax Score” represent a rating out of 5 of the AI skill levels provided by the players.

References

Brown, M. (2020). The Two Types of Random in Game Design. YouTube. Available at: https://www.youtube.com/watch?v=dwI5b-wRLic [Accessed 24 Apr. 2021].

Churchill, D. and Buro, M. (2017). Hierarchical Portfolio Search in Prismata. In: S. Rabin, ed., Game AI Pro 3: Collected Wisdom of Game AI Professionals. Taylor & Francis Group.

Cowling, P.I., Ward, C.D. and Powley, E.J. (2012). Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. IEEE Transactions on Computational Intelligence and AI in Games, 4(4), pp.241–257. doi:https://doi.org/10.1109/tciaig.2012.2204883.

Doyen, L. and Raskin, J.-F. (2011). Games with Imperfect Information: Theory and Algorithms. [online] Available at: HYPERLINK "https://lsv.ens-paris-saclay.fr/~doyen/papers/Games_with_Imperfect_Information_Theory_Algorithms.pdf"https://lsv.ens-paris-saclay.fr/~doyen/papers/Games_with_Imperfect_Information_Theory_Algorithms.pdf.

Edwards, G., Subianto, N., Englund, D., Goh, J.W., Coughran, N., Milton, Z., Mirnateghi, N. and Ali Shah, S.A. (2021). The Role of Machine Learning in Game Development Domain - A Review of Current Trends and Future Directions. 2021 Digital Image Computing: Techniques and Applications (DICTA). doi:HYPERLINK "https://doi.org/10.1109/dicta52665.2021.9647261"https://doi.org/10.1109/dicta52665.2021.9647261.

Hartley, T. and Mehdi, Q. (2011). In-game tactic adaptation for interactive computer games. doi:HYPERLINK "https://doi.org/10.1109/cgames.2011.6000358"https://doi.org/10.1109/cgames.2011.6000358 .

Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M. and Bowling, M. (2017). DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker. Science, [online] 356(6337), pp.508–513. doi:https://doi.org/10.1126/science.aam6960.

Mozgovoy, M. (2018). Context-Awareness and Anticipation in a Tennis Video Game AI System. 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC), [online] pp.699–703. doi:HYPERLINK "https://doi.org/10.1109/smc.2018.00127"https://doi.org/10.1109/smc.2018.00127 .

Rabin, S. (2017). Game AI Pro 3 : collected wisdom of game AI Professionals. Boca Raton, Fl: Crc Press, Taylor & Francis Group.

Ramlan, A.A.B., Ali, A.M., Hamid, N.H.A. and Osman, R. (2022). The Implementation of Reinforcement Learning Algorithm for AI Bot in Fighting Video Game. [online] International Symposium on Agents, Multi-Agent Systems and Robotics. Available at: HYPERLINK "https://ieeexplore.ieee.org/document/9567749"https://ieeexplore.ieee.org/document/9567749.

Roelofs, G.-J. (2017). Pitfalls and Solutions When Using Monte Carlo Tree Search for Strategy and Tactical Games. In: S. Rabin, ed., Game AI Pro 3 : Collected Wisdom of Game AI Professionals. Taylor & Francis Group.

Shvets, A. (2014). Strategy. [online] Refactoring.guru. Available at: HYPERLINK "https://refactoring.guru/design-patterns/strategy"https://refactoring.guru/design-patterns/strategy.

Spronck, P.H.M. (2005). Adaptive game AI. [online] doi: HYPERLINK "https://doi.org/10.26481/dis.20050520ps"https://doi.org/10.26481/dis.20050520ps.

Sturtevant, N. (2015). Monte Carlo Tree Search and Related Algorithms for Games. In: S. Rabin, ed., Game AI Pro 2: Collected Wisdom of Game AI Professionals. Taylor & Francis Group.

Watanabe, R., Araswa, K. and Hattori, S. (2018). Rule-Based Role Analysis of Game Characters Using Tags about Characteristics for Strategy Estimation by Game AI. In: International Conference on Soft Computing and Intelligent Systems and 19th International Symposium on Advanced Intelligent Systems. [online] Available at:HYPERLINK "https://www.e.usp.ac.jp/~hattori.s/pdf/watanabe2018isws.pdf" https://www.e.usp.ac.jp/~hattori.s/pdf/watanabe2018isws.pdf [Accessed 19 Nov. 2025].

Wetzel, S., Spiel, K. and Bertel, S. (2014). Dynamically adapting an AI game engine based on players' eye movements and strategies. doi:HYPERLINK

"https://doi.org/10.1145/2607023.2607029"https://doi.org/10.1145/2607023.2607029.