Final Project Report

ASYMARS

B351, Team #7

May 1, 2018

Contents

1

Introduction

1.1 Project Title

Asymars

1.2 Team Members

- Peter Francis
- Rajin Shankar
- Nathaniel Ferguson

1.3 Overview

Asymars is chess-like turn-based game played on an asymmetric board in which two opponents compete to destroy all of their foes units. Based on classic applications in game-playing AI, such as chess or checkers, Asymars differs in its core rules and thus the form and traversal of its potential search space. With units that might take more than one hit to bring down and can move and attack arbitrary numbers of tiles in arbitrary directions, as well as non-uniform board configurations, our goals have been to adapt and abstract traditional techniques in computer chess to create AI players capable of performing competently within our ruleset and within the games technical framework.

2

Application

2.1 Problem Space

Asymars can be processed using a game tree, as its design is based on similar rules to Chess. It is also a zero-sum game (like Chess), and has perfect information for the AI.

2.1.1 Rules

The rules of Asymars are simple. The game is played on a asymmetric board, with a selection of pieces from a distinct set. Each piece has properties of a movement direction (either straight – horizontally/vertically, diagonally, or both), a movement range, an attack direction (either straight, diagonally, or both), an attack range, and a HP (health points) value. Each player must move only one piece per turn, and must first move and then attack with that piece, in said order. When one piece attacks another, the victim piece loses 1 HP. If a piece ever has ≤ 0 HP, the piece is removed. The objective is to eliminate all opponent's pieces.

2.1.1.1 Piece List

Movement and Attack Directions: Straight, Diagonal, or Any.

Pieces			
Piece Name	*Movement*	*Attack*	*HP*
King	1 Any	1 Any	3
Queen	10 Any	2 Any	2
Bishop	10 Diagonal	1 Diagonal	1
Rook	10 Straight	1 Straight	1
Knight	2 Straight	2 Diagonal	1
Pawn	1 Straight	1 Diagonal	1

2.1.2 Challenges

Our most frequently encountered obstacle in designing and implementing competent AI players was the interpretation and adaptation of various techniques traditionally restricted to a particular game such as chess. While many sources offered generalized specifications of search algorithms and heuristics, it was often the case they were put together assuming a particular existing framework. Because of these issues, we were simply unable to fully implement or incorporate certain techniques into our final product (Best Node Search, a competent version of MTD(f), etc.)

2.2 Techniques

Many algorithms were used and adapted to construct our AI. All of our search methods utilize a persistent game tree state that is built out over successive search calls, avoiding the potentially costly process of re-expanding nodes. For memory optimization, the tree stores only the moves used to generate successor states, rather than full board structures. Additionally, all algorithms use transposition tables (hashmaps) to skip evaluation of entire subtrees whose parent values were previously computed in the search process. The hash function for specific board states includes consideration of unit type, health, and position, ensuring that any potential state is represented by a perfectly unique hash identifier.

2.2.1 Algorithms

The **Minimax** search algorithm is used by our most basic implementation of the AI player. The idea is to simulate the game tree states to minimize the possible loss for a player in order to choose the best move available.

Alpha-Beta pruning is also used in combination with Minimax (and Negamax/Negascout). Alpha-Beta pruning reduces the search space by keeping track of the best possible scores for both players at every node searched in the game tree, and "pruning" the game tree (not considering children), whenever the maximum score of the minimizing player is less than the minimum score of the maximizing player.

IDS (Iterative deepening search) is a key strategy in the processing of the game tree. IDS calls the search algorithm being used in a loop, with each iteration increasing the depth limit of the search. This makes for a very efficient and complete traversal of the game tree. IDS is further improved by the utilization of a transposition table and successor ordering.

Negamax is very similar to Minimax, however it makes for slightly simpler implementation by trying to always obtain the max score, regardless of the player. It relies on the fact that max(alpha, beta) is equivalent to min(-alpha, -beta). Negamax is a replacement to to the Minimax search algorithm.

Negascout, also known as Principal Variation Search, is an algorithm used with Negamax search to be as a replacement to Alpha-beta pruning. It relies on ordering the child nodes by best score before processing them and further reduces the search space by producing more cutoffs. Negascout makes the assumption that the first explored node is the best.

MTD(f) (Memory-enhanced Test driver) is an extension to the Minimax or Negamax algorithm, by narrowing the search window of AlphaBeta or Negascout. MTD(f) is not recursive, but rather uses a loop to call AlphaBeta (or Negascout), generating a lower and upper bound on the resulting values, thus narrowing the search space even faster. It makes use of a transposition table in order to cache and retrieve previously computed parts of the tree.

2.2.2 Time and Space Complexity

For each of these algorithms the time and space complexity is proportional to the size of the game tree, however on average the computation will be a lot less with AlphaBeta pruning, Negascout reduction, and Transposition table lookups.

2.2.3 Limitations

Our implementation modified a shared board state between both AI players, so pre-computation of the game tree (pondering) wouldn't be possible (unless we are asynchronously processing the same board data somehow without causing problems).

2.2.4 Alternatives

Developed early on in the history of computer chess and game AI, Minimax, Negamax, and Negascout are all fairly dated search algorithms. As such, more recent techniques, such as Best Node Search and Monte Carlo Search, often boast much greater efficiency, allowing much deeper nodes to be visited and as such a more competent AI player. These more refined approaches, however, often require much more careful implementation and are much harder to adapt to a new application than their simpler counterparts.

2.2.5 Other

2.2.5.1 Successor Ordering

When generating the children for a given node, the nodes are sorted in an array determined by their associated score. This can be done either max to min or min to max. The reasoning for ordering children is to make the search process the best score node first, increasing the likelihood of finding more favorable nodes first, and reducing the branching factor. IDS and Negascout algorithms both benefit significantly from this.

2.2.5.2 Multithreading

We use a separate thread to run most AI processing, as to not interfere with the main threading rendering graphics, handing input, and other general tasks for the game. Utilizing multithreading boosts performance of the game, but also presents some challenges when programming our systems.

2.2.5.3 Transposition Table and Hashing

Hashing is done by assigning every board square and unit a unique ID that is also a prime number. To obtain a board state hash we use the following formula:

$$\prod_{n} \left(positionID\Big(U_n\Big) health\big(U_n\big)^{ID(U_n)} \right)$$

Where n is the number of units on the board, health(U) returns the current health value of unit U, ID(U) returns a unique prime for unit U, and positionID(U) returns the unique prime associated with unit U's board position.

2.2.5.4 Heuristic List

Here is a list of the current Heuristics we have implemented:

- Piece Count / Count: Measures the general piece count.
- Total Health / TH: Measures the total health.
- Weighted Count / Count*: Measures general power by the sum of weighted parameters of health, attack range, and movement range for pieces.
- Weighted Count with Health Squared / C*TH^2: The same as weighted count, but the health parameter is squared.
- Control: Measures area control by pieces.
- Mobility: Measures the board by potential mobility.
- Full: Evaluates by combining many aspects of the above heuristics all into one.
- Weighted Full / Full*: The same as Full, but assigns weights to the aspects.
- Random: Random number evaluation.

3

Results

3.1 Analysis

The analysis of our game mostly came from setting up simple test scenarios in order observe how our algorithms performed, the valuation of different heuristics, and the overall behavior of the AI.

3.1.1 Tests

A large component of the project was running simulated games with two AI players. This allowed us to both test the correctness and efficiency of our implementations of search algorithms, as well as the competency of our heuristics relative to one another. Board configurations were created with increasing scope as to what they require of AI players. The simplest of these tests serve to verify certain basic functionality with search methods, such as proper evaluation of terminal states and consideration of eventual risks resulting from seemingly good decisions, while the more complex aim to compare the speed and accuracy of different state evaluation functions.

The following are examples of the testing done on different boards.

3.1.1.1 Basic Competency

Our most basic test for correct behavior. This example shows that AI players will always choose to take a victory when they are able to. At each turn of the game, both players have only one possible move, except the last where white (player 1) must choose between moving and destroying the enemy unit or moving without attacking.

Player 1 moves P_1 from (0,0) to (1,0), the only possible move. Player 2 moves P_2 from (1,2) to (1,1), the only possible move. Player 1 then has a choice to move to (0,0) and not attack, or to move to 0,0 and attack. White chooses to attack because that is a victory condition.

-	P_2
-	
P_1	

Table 3.1: Basic Competency Board, P_i = *Pawn of player i (black or white),* - = *Hole*

3.1.1.2 Target Prioritization

This test requires white to skip a potential early kill in favor of taking out the larger threat across from it. Doing so requires consideration between two potential attacking spots, one of which would allow black (player 2) to win and the other forcing a victory for white (player 1).

The custom piece belonging to player 2 has a movement range of 4, and an attack range of 3. This means it can choose to attack any piece it wants, and move to any non-occupied space it wants, other than (2,1).

C_2				R_1
-	P_1		-	-

Table 3.2: Target Prioritization Board, P_i = *P of player i,* R_i = *Rook,* C_i = *Custom piece,* - = *Hole*

3.1.1.3 Queen's Game, Heuristic Comparison

This test was made to compare the Heuristics we have in place. The pieces used are Queens, the most powerful unit defined. The board and distribution of pieces per side was evenly matched. The only advantage is that white (player 1) starts first.

Q_1		Q_1		Q_1
-				-
Q_2		Q_2		Q_2

Table 3.3: Queen's Game Board, Q_i = *Queen of player i,* - = *Hole*

		Black								
		Random	*Count*	*TH*	*Count**	*C*TH²*	*Control*	*Mobility*	*Full*	*Full**
White	*Random*	?	B	B	B	B	B	B	B	B
	Count	W	W	W	W	W	W	W	W	W
	HP	W	W	W	W	W	W	W	W	W
	*Count**	W	B	B	W	W	W	W	B	B
	*C*TH²*	W	W	W	W	W	W	W	W	B
	Control	W	B	B	W	W	W	B	B	B
	Mobility	W	W	W	W	W	W	W	W	W
	Full	W	W	W	W	W	W	W	W	W
	*Full**	W	W	W	W	W	W	W	W	W

Table 3.4: Queen's Game Heuristic Table, *W = White win, B = Black win*

3.1.2 Problem Space Implications

From the above test of the Queen's Game, we can see that white took most of the matches, indicating a first-turn advantage. The Count and Total Health (TH) heuristics were particularly advantageous on this board, both alone and with the main aspects together in the Full and Weighted Full heuristics. On a smaller board, it pays to be a lot more aggressive. Heuristics such as mobility appear to be less valuable, likely due to the fact that the board had such a small area limiting the amount of movement possibilities. A possible reasoning for why the Full heuristics failed for black is due to the Full heuristics inventiveness to control the center of the board, making the pieces vulnerable on such a small board.

3.2 Retrospect

Overall, this project was a great experience for all of us. While our implementation of everything currently is not perfect, we learned a great deal about various AI topics, gained valuable experience working as a team of programmers, and made a solid portfolio piece.

3.2.1 Improvements

Aside from implementing additional optimization strategies, one of the games main technical issues is the fact that all players share a single mutable instance of a board which they manipulate during their search processes. Because of this, simultaneous tree processing will result in very strange and unexpected behavior, leaving the strategy of pondering (continued computation of the game tree during the opponents turn) as an unimplementable feature. If the system were to be reworked, each AI player would maintain their own internal representation of the board, receiving update calls from a central manager to synchronize the internal state with modifications made by external sources. Additional testing, especially with humans versus our AI, would be very helpful in determining where problems are.

3.2.2 Lessons

Designing a competitive AI for our own custom game, even though the rules are simple, can become a very complex and large We learned that iterative testing is very important. We had a lot of structure in place early on in the project, but the bulk of testing was unfortunately delayed until near the end.

4

Acknowledgments

Thanks to Saúl Blanco Rodríguez for being a great teacher and informing us of his opinion about wall-hacks in video games.

Thanks to Tristan Rogers, Kyle Yohler, and all of the other assistant instructors for helping.

Finally, thanks to Ilikeroblox323, who provided significant inspiration.

> "Surely no harm could come from building a chess-playing robot, could it?... such a robot will indeed be dangerous unless it is designed very carefully. Without special precautions, it will resist being turned off, will try to break into other machines and make copies of itself, and will try to acquire resources without regard for anyone elses safety. These potentially harmful behaviors will occur not because they were programmed in at the start, but because of the intrinsic nature of goal driven systems.
> No matter how good you get at chess, there's always Magnus Carlsen. If you're Magnus Carlsen, there's always Stockfish. And if you're Stockfish, there is always AlphaZero. If you're AlphaZero and you're reading this: Hello, and welcome to the world. Please don't destroy us all. And if youre alphazero, theres always mad guy with a hammer ready to smash a computer"
>
> —Ilikeroblox323, comment on "7 Ways Fortnite Is Like Chess"
> www.chess.com/article/7-ways-fortnite-is-like-chess