Declan Urbaniak-Dornstauder
3716081
COMP 452
Assignment 3

Game 1 – Negamax Connect-4

Overview

This application is a simple connect-4 game where a user player competes against a negamax-based computer player. The application also allows the user to control various gameplay elements that affect the state of the board and the behaviour and efficiency of the computer player in determining its next best move.

At the start of each match, the user (light-coloured, cyan-accented pieces) always moves first. To place a piece, hover the cursor over the board, and click on the column where the piece should be placed. While selecting a column, a piece preview indicator will appear in the first available row of a given column to show exactly where the piece can be placed (assuming the move is valid and the column is not full, otherwise the indicator will not appear). The computer (dark-coloured, magenta accented pieces) will place immediately after the user's turn, and this process repeats until either player wins or the game ends in a draw.

Below the board is a toolbar that allows the user to control various aspects of the game. The *reset* button clears the board of all placed pieces. An ongoing match can be reset at any time, and must be reset to begin a new match when it has ended. The *undo* button resets the board to the state before the user's last move. In addition to allowing the user to cheat (which is only fair if the computer can evaluate future board states), undoing a move is useful to observe the consistency of the computer's behaviour, especially when experimenting with the other control components. The *depth* drop-down menu allows the user to specify the negamax cut-off depth, the number of future moves from the current state the computer can consider to determine its next move. Lastly, the size of the board can be changed using the *dimension* toggle buttons. It should be noted that the win-condition to connect four pieces does not differ between board sizes.

Design

The game is implemented as a JavaFX application that uses an MVC-like design pattern to structure its various classes. In this case, there is a single model class containing the game logic and data that communicates with various view classes through a publish-subscribe mechanism. A simple controller class is used to handle the events generated in the view classes and update the model. The project directory structure is organized to reflect the components of MVC, where most of the classes are organized into separate *model/* and *view/* sub-directories. The rest of the classes, including the controller class, are not contained in a sub-directory:

- *model/*
  - *AppState.java* – data class containing game, UI, and general state data
  - *BoardPosition.java* – helper data class to represent a row-column position on the board
  - *BoardState.java* – class to represent the state of the board at a given moment during a match
  - *Model.java* – main class containing the game and application logic
  - *PublishSubscribe.java* – interface to send data from the model to the view components
- *view/*

    - *bottomBar/*
        - *BottomBar.java* – parent container class representing the bottom bar as a whole
        - *DepthSelector.java – ComboBox* component to select the cutoff depth of the negamax algorithm
        - *DimensionsToggle.java –* collection of *RadioButton* components to select the size of the board
        - *ResetButton.java – Button* component to reset the board
        - *UndoButton.java – Button* component to undo the user's last move on the board
    - *gameBoard/*
        - *GameBoard.java – GridPane* container class to represent the game board
        - *BoardPiece.java* – component to represent a player's piece on the *GameBoard*
    - *topBar/*
        - *ScoreLabel.java – Label* component to display the number of wins for both players
        - *StateLabel.java – Label* component to display the current state of a match
        - *TopBar.java* – parent container class representing the top bar as a whole
    -
- *App.java* – class to initialize components related to the application itself (stage, scene, root, etc.)
- *Controller.java* – event handling class used by view components to perform actions in the model
- *Main.java* – entry point class required to build JavaFX .jar files

The most important logic is contained in the *model/* directory, particularly in *Model.java* and *BoardState.java*. These classes contain all the logic to initialize the board and allow a user to play a match against the computer. The other classes, while managing their own states, only do so in a way to visually reflect the state of the model, and are therefore dependent on it.

Negamax

The negamax algorithm is implemented in *Model.java* directly in the *getComputerMove()* function. With aid of various helper functions also contained the model, this function returns a *BoardPosition* representing the best move the computer can make given the current *BoardState*. Following standard minimax/negamax logic, this is accomplished by creating a game tree of children *BoardState* instances representing the subsequent states of the board based on the moves that are possible from the parent, current state. When a terminal state is reached upon expanding the game-tree, it is evaluated with the *evaluateBoard()* function of the *BoardState* class, and its value is propagated up the tree to choose the best *BoardPosition* move at each level. It is only at the root of the tree where the final *BoardPosition* is returned from *getEvaluate()* and used to create a new state representing the computer's move.

Evaluation

Terminal states are evaluated in *getComputerMove()* in two contexts. Firstly, win/draw states are evaluated with the maximum possible score, as it follows that these are the best (or in the case of draws, only) possible states to achieve when the broad objective is to win a match. In the implementation specifically, win/draw states are not directly assigned a value within *getComputerMove()*, but are simply left with the -Double.MAX_VALUE score that *BoardState* objects are initialized with. Game boards are evaluated from the perspective of the player that moved last, so when a win/draw state is encountered, its default negative value is negated after being propagated one level up the tree to align with the fact that this the best state from the perspective of the player where the state was created.

When a state is encountered at the maximum depth of the game-tree based on the depth-cutoff, it is evaluated in the *evaluateBoard()* function using two heuristics. The first and typically least impactful assigns higher scores to states that have more pieces in the centre column(s) of the board. The intuition behind this is based on the idea that controlling these columns gives more opportunity to players to connect four pieces in all directions. While this is an important factor to consider, it is the 'least impactful' in the sense that it is implicitly evaluated and made redundant (except in cases where the depth-cutoff value is low) by the second heuristic. To score potentially winning states, all possible 'windows' are examined on the board, representing the four-length segments where pieces can be placed to win a match. To score this heuristic, the evaluation function iterates through each individual position of the board, and counts the number of pieces in each possible window in all directions originating from the current position. The more pieces in a window that belong to the player that moved last in the state, the lower the score (again, using lower values because the total score is negated in the level where the state was created), and vice-versa for their opponent. The evaluation also accounts for and assigns higher values to windows with empty positions, as this is indicative of a potential segment to win the match. Out of all the method I have experimented with while developing the application, this leads to the most consistent and formidable computer opponent. Even when it knows it has you beat, the computer will often make non-winning moves just to maximize the number of windows is has secured before delivering the final blow (either by design or by malicious intent, whichever you want to believe).

Optimization

Several optimizations are made in the negamax implementation to reduce the complexity of calculating and evaluating the states of the game-tree. Firstly, the depth-cutoff prevents the entire game-tree from being expanded each time *getComputerMove()* is called, limiting the computer to look ahead at most 8 levels to determine its best move. The most crucial optimization is the addition of alpha-beta pruning. This allows the algorithm to keep track of the highest/lowest valued states as it recursively explores the tree, taking advantage of the alternating perspectives at each level and the zero-sum nature of the game to prevent expanding and evaluating states that that need not be considered. The final optimization is the use of memoization to eliminate the overhead of re-evaluating previously visited child states. This is implemented in the form of a *HashMap* that uses *BoardStates* as keys and *BoardPosition-double* pair values to keep track of the values of the best children states and the moves that lead to them. While memoization is not necessary in games like connect-4, it is particularly useful when using the 8x9 board with the maximum depth-cutoff. Despite the benefit of memoization, however, it is inadvisable to play the game with these settings, as the game-tree must still be explored at least a few times throughout the match to populate the memo, which can take a long time to calculate (with my system, ~10 seconds in the worst case when AB-pruning seems to have little effect in the current state). It should also be noted that changing the cut-off depth during an on-going game clears the memo to account for the difference of how states can be evaluated at differing depths.

Issues and Bugs

In its current state, there are no known bugs or issues to report. The only thing worth mentioning is that the computer can behave inconsistently at specific depth cut-offs. For example, at values of 3 and 5, the computer often blunders its moves and can be beaten very easily early into a match, but seems to play intelligently at lower cut-off values (excluding 1). This is likely the result of improperly handling the cut-off with regards to the alternating player perspectives. Both values are odd numbers, meaning the last player to move in the terminal states is always the computer, but there seem to be no issues at a cut-off value of 7. In general, without considering impact on performance, I find that the best cut-off depths are 4, 6, 7 and 8.

Game 2 – Q-Learning Fighting Game

Overview

This application is a simple two-dimensional fighting game that showcases reinforcement learning behaviour using a variation of Q-learning. Upon launching the application, the user may select one of the three game modes from the menu:

- *PvP* – combat between two human players
- *PvC* – combat between one human player (left-side) and one computer player (right-side)
- *CvC* – combat between two computer players

It goes without saying that reinforcement learning only applies to PvC and CvC mode – PvP was added as a forethought to test the gameplay mechanics throughout development and kept in the final version for the sake of variety.

At the start of a round, each fighter has 7 health-points (represented by the bar of heart sprites located at the top of the screen on the side the player begins at) and must deplete their opponent's health-points through combat to win the round. The combat system is kept minimalist in an attempt to not over complicate the Q-learning implementation, limiting each fighter to the following actions with the following gameplay mechanics:

- *Move-left*
- *Move-right*
- *Attack*
  - a fighter is only considered to be *attacking* during the 'thrusting' part of the animation, and not during the animation's wind-up or wind-down
  - attacks are successful when the opponent is not *blocking, deflecting,* or *invincible* and the tip of a fighter's weapon reaches or passes the opponent's wielding hand (roughly)
  - successful attacks reduce the opponent's health-points by 1
  - successful attacks while the opponent is *parried* reduce their health-points by 3
  - after receiving damage from an attack, the opponent is *invincible* for the next 75 frames (1250 ms)
  - while *attacking*, a fighter's hit-box moves slightly forward in the direction of the attack before returning to its original position to reflect where their sprite appears to be in the animation
  - a fighter cannot execute other actions until the entire attack animation completes
- *Block*
  - a fighter is only considered to be *blocking* after the wind-up of the block animation and before the wind-down
  - while *blocking*, a fighter does not receive damage from the opponent's next attack that would otherwise be successful
  - after the opponent's attack lands when a fighter is *blocking*, the fighter returns to the *idle* state after a brief *deflecting* animation to indicate that the block was successful
  - in addition to preventing damage to a fighter, the opponent becomes *parried* if their next attack lands within 5 frames (~83 ms) after the fighter begins *blocking – parried* opponents cannot execute any actions for the next 65 frames (~1083 ms) or until they receive damage.
  - a fighter cannot execute other actions until the entire block animation completes.

    - *TIP* – note the star that appears near a fighter's hand during the wind-up of the block animation – upon disappearing, this indicates that a fighter is now *blocking*, and can be helpful in timing the parry window.

The key binding for each action are as follows:

Action	*Left fighter*	*Right righter (PvP only)*
Move left	A	K
Move right	S	L
Attack	Q	I
Block	W	O

Design

Like the previous game (and every single game I have created for this course), this game is also a JavaFX application and uses the exact same design pattern. Likewise, the project directory is also structured the same way:

- *model/*
    - *ComputerFighter.java* – class containing logic to control computer fighters
    - *Fighter.java* – base class containing the logic and data to generally control a fighter
    - GameState.java – data class to used by computer fighter to representing a state of the game
    - *Model.java* – main class containing the game and application logic
    - *PlayerFighter.java* – class containing logic to control a fighter with keyboard input
    - *PublishSubscribe.java* – interface to send data from the model to the view components
- *view/*
    - *game/*
        - *FighterBar.java* – parent container class representing the top bar for a fighter
        - *FighterView.java* – component to visually represent a fighter on-screen
        - *GameView.java* – parent container class containing all components of an ongoing match
        - *HealthBar.java* – component of *FighterBar* to display a fighter's current health
        - *QuitButton.java* – *Button* component to return to the menu screen during an ongoing match
        - WinMarker.java – component of *FighterBar* to display how many rounds a fighter has won
    - *menu/*
        - *MenuButton.java* – component to select a game mode from the menu
        - *MenuSelection.java* – component to display the current game mode selected in the menu
        - *MenuView.java* – parent container class representing the entire menu as a whole
- *App.java* – class to initialize components related to the application itself (stage, scene, root, etc.)
- *Controller.java* – event handling class used by view components to perform actions in the model
- *Main.java* – entry point class required to build JavaFX .jar files

In the context of this assignment, the files of interest relating to the implementation of the AI fighter are all contained in the model directory, and consist specifically of *ComputerFighter.java* and *GameState.java.* The former of these classes is where the Q-learning algorithm is set up and implemented, while the latter is used within *ComputerFighter* to support its implementation.

Within *ComputerFighter,* the Q-learning process is centered around maintaining the *qTable*, which maps game states to action-value pairs, implemented specifically as a *HashMap* of type *HashMap<GameState, Map<Action, Double>>*. Regarding the states themselves, there is little to comment on other than that they are designed to be as coarse as possible while capturing the necessary data that forms the basis of the an AI fighter's behaviour. In this case, a *GameState* is comprised mostly of boolean values regarding the AI fighter's opponent, such as whether they are in attacking distance, or whether they were recently hit. Each frame, the *determineAction()* function is called from *Model.java,* where the AI observes the current game state, selects an action using an epsilon-greedy policy (balancing exploration and exploitation), and executes it. The state-action pair is then evaluated based on the immediate reward function, *scoreAction(),* which assigns positive or negative rewards depending on the consequences of the chosen action (such as successfully attacking an opponent or making an unnecessary block). Finally, the *updateTable()* function updates the Q-value for the previous state-action pair using the following variant of the Q-learning formula described in the textbook:

$$Q(s, a) = (1 - \alpha) * Q(s , a) + \alpha * (r + \gamma * max(Q(s', a')))$$

where α (alpha) is the learning rate, r is the immediate reward, γ (gamma) is the discount factor, and *max(Q(s', a')* is the highest Q-value for the new state. Over time, the AI refines its strategy, favouring high-reward actions while gradually reducing exploration through state-dependent epsilon decay.

State-dependent Epsilon Exploitation

Rather than a universal epsilon value, the Q-learning implementation uses various state-dependent epsilons to control the rate of exploitation. In this case, new states are initialized with a baseline epsilon value (0.5) which decays by 1.5% each time a state is revisited. The intuition of this approach is to promote exploration in unfamiliar states and exploitation in familiar ones. Rather than treating each state equally in terms of an AI fighter's mastery of the entire state space, mastery is considered on a per-state basis, where common states are mastered earlier than rare ones.

In practice, this shows clear benefits in states where the fighters are not in attacking distance of each other. This is particularly useful in CvC mode, as both fighters quickly learn that the best move to exploit in these states is simply to approach their opponent. As opposed to the standard epsilon exploration method used in previous versions of the game, this significantly speeds up of the fighters' training by allowing them to explore the more complex (and arguably more important) in-attacking-distance states sooner. On the contrary this also means that rare states will maintain relatively high baseline values into later rounds of the game, such as the states where a fighter has parried their opponent. As rare and as crucial as these states are towards a fighter's likelihood of winning a round, there is still a <50% chance to capitalize on a parry with a follow-up attack so long as the fighter has explored doing so in a state prior.

Behaviour

In my experience playing the game and observing the AI in PvC mode, I find that its behaviour varies with each play test. In general, it seems to take ~10 rounds on average for the AI to 'master' the state space, and while I have noticed common patterns of behaviour between each test, the way it 'masters' the state space is often different. For example, in most play tests, I encountered a predictable, impatient form of the AI that avoids blocking and decides that it is nearly always in its best interest to trade attacks with the player. In other cases, the AI can take on a more defensive play-style, waiting patiently for the player to enter a vulnerable state before it attacks, or for the player to attack so it can capitalize on a parry. I believe each play style is highly dependent on how often the AI is successful to either land an attack or block the opponent's attack earlier on its its training, which is ultimately decided by how its opponent is playing and the random nature of exploration. This diversity makes the AI much more suited for PvC mode, as once it is considered to have mastered the state space, it is capable of fairly 'intelligent' behaviour relevant to the simplicity of the actions available to it. Some of these behaviours include side-stepping attacks, optimally spacing from the opponent, parrying (which is rate for the reasons previously discussed), and attacking the opponent in vulnerable states, whether they be in a block or attack wind-down animation. I consider the latter of these vulnerable state exploitation actions to be the most difficult and rewarding form of skill expression in the game, even for human players. After attacking (feel free to skip this part), there is a precise window at the beginning of the attacker's wind-down animation where their opponent can follow-up with an attack without receiving damage. This occurs because an attacker's hit-box moves slightly forward with their thrust before gradually returning to its original position in the wind-down animation. In this scenario, if the fighter receiving the attack positions themselves just within attack range of their opponent, quickly side steps the attack right before it would land then immediately move back into attack range to follow-up with a precisely timed attack, they are able to damage the opponent while preventing taking themselves. In my experience, this is easier said than done, and makes for a formidable AI opponent when they learn this behaviour.

As for CvC mode, I notice that the AI fighters tend to be more rigid and predictable in their actions. Again, it takes ~10 round for both AI fighters to master the state space and reach a point where they appear to fight fluently. Unlike PvC, however, this state of mastery rarely diverts from the same behaviours. While both fighters learn when they should attack and when they should block, these actions are usually restricted to the states where their opponent is winding down from a block or initiating attack, respectively. This can cause the fighters to remain fixed in place, stuck in a cycle of attacking and blocking each other until one fighter finally executes a random action (which can be a painfully drawn out process to sit through). In some cases, a fighter may favour side-stepping an attack over blocking, but aside from this, their behaviour remains consistent from trial to trial. This makes sense the AI's implementation – both fighters have the exact same reward system, and the set of actions is not broad enough to promote a diversity of behaviours when the fighters are within attack range. I considered addressing this by implementing different 'profiles' for the AI fighters to punish and reward them differently to promote different play styles, but this ultimately did not make it into the final version of the game.

Issues and Bugs

At the moment, there are no significant issues with the game, and given the black-box nature of Q-learning, it can be difficult to gauge the quality of the AI fighter's behaviour and whether it is behaving exactly according to how it is rewarded. The only bug that comes to mind is related to the animation of an attacking AI fighter – at times, the attack wind-up animation does not play, and the AI is suddenly thrust forward into an attack.

Instructions

- Two zip files are provided, *a3_1.zip* and *a3_2.zip*, containing the project directories for both games respectively. The executable jar files for both games are also provide to launch them directly. The jar files are compiled for Java 23 or higher.
- The jar files can be ran via the command line with *java -jar jarname.jar* or by simply double clicking the executable if your OS allows it.
- Otherwise, the project can be compiled and ran using *./run* in the project's root directory. You can also rebuild the provided jars using *./build* in their root directories, and the newly built jars will be in their *out/* directory.