L Game Report

Group: Raghav Sinha, Brandon Wong
Contribution: Raghav: 50%, Brandon: 50%

Design Choices

- We chose to represent the game state using a 2d list of the board itself and nothing else. The board contained numbers to specify empty positions, L positions, and neutral piece positions. We chose to consider these 2d lists as units across the code, not using another universal data structure to represent any other part of the game. This was because all of the information needed at any point to describe the game lives within the board itself, so there is no need to create any further data structures. Creating further data structures gives the chance of there being disconnects between the different methods of representing data.
- We chose to represent legal actions as states themselves, not as “moves”. This was to keep things consistent and not introduce too many types of data and have to convert between multiple times in any subroutine. Every legal successor state was a state, not a legal move.
- We chose to break down the L game into these functions and their purposes. Our philosophy here was compartmentalization of code - if a subroutine can be created, it will be created. We also created some functions to help readability for our larger functions:
    - menu()
        - Our menu includes options to play pvp, pvc, and cvc, as well as choosing if the player or cpu starts, changing and resetting the initial state, and quitting the game. We gave players the option of inputting the number assigned to the option, or the abbreviation of the option.
    - isValidInitialState(initState)
        - This function checks if the player’s manually entered initial state is valid, checking both its length and formatting. We needed to check that the input coordinates of the Ls and dots were valid, not overlapping over the other or out of bounds. While not a subroutine, it helps with the readability of the changeInitialState function.
    - changeInitialState()
        - This function is called when the player chooses to change the starting board of the game. It asks the player to enter a new initial state and calls isValidInitialState(initStateInput) in order to check if the entered initial state is valid, as well as gives the player the option to quit entering a new initial state entirely. The inputted initial state is used to generate L coordinates by calling generateLCoords, as well as assigning both dots. The L and dot coordinates are then mapped to the initial state board and the initial state board is returned.
    - generateLCoords(x, y, direction)
        - This function is a utility function that is only used in isValidMove. It generates a list of coordinates(tuples) for the L given the corner

coordinate and the direction of the L. Though this isn't a subroutine, it helps with readability of the isValidMove function.

- playGamePVP()
    - This function handles the entire Player versus Player game scenario, printing out the board, accepting both players moves, and checking for win/loss scenarios. While not a subroutine, it helps with the readability of our menu function.
- playGamePVC(agent)
    - This function handles the entire Player versus computer game scenario and takes in which agent will be considered the player (blue or red). It prints out the board, accepts the player's moves, generates the computers moves by calling our AI function, and checks for win/loss scenarios. While not a subroutine, it helps with the readability of our menu function.
- playGameCVC()
    - This function handles the entire computer versus computer game scenario. It prints out the board, generates the computers' moves by calling our AI function, and checks for win/loss scenarios. While not a subroutine, it helps with the readability of our menu function.
- printBoard(board)
    - This utility function takes in a gameState as "board" and prints it out in an nice way, using colors to represent the different L pieces for appeal and readability. It is a very useful subroutine used throughout the "play game" logic and was very useful for debugging to visualize the current state at any moment.
- applyMove(board, move, agent)
    - This function takes in a gameState as "board", as well as the current agent as "agent" and orientations as "move". The function iterates through move to change the coordinates of the L based on the direction of L and the movement of the dots.
- dotProximityToOpponent(dotPositions, opponentLCoords)
    - This is a utility function only used in our heuristic. It returns the proximity of each neutral piece to each of the given opponent L coordinates. We made this to help with readability. This function is probably the most deletable of the bunch, but it helps us read our heuristic, so we kept it.
- evaluateAction(nextGameState, agent)
    - This was our heuristic function, taking in the next game state and the current agent and returning a heuristic value evaluating the next game state for its positive or negative value to the current agent to achieve the win scenario of there being no moves for the opponent. We will go into more detail about our heuristic later in this report.
- isStateInStates(state, stateList)

    - This function checks for a given state if it is in a given list of states. This very general and simple action was used a lot throughout the code, so we made it into a utility function to handle the subroutine.
- findStateInStates(state, stateList)
    - This function, similar to the last, finds a state in a stateList and returns the index in the list if found or -1 if not found. This general action was also used a lot in our algorithm, so we made it into a utility function to handle the subroutine.
- compareStates(state1, state2)
    - This function compares two given states and returns true if they are exactly the same. This is a useful function because it is a further subroutine in our already defined functions isStateInStates and findStateInStates, and we use it in other places in the code as well, further validating it as a subroutine.
- getBestSuccessor(gameState, agent)
    - This function is our AI function. When this function is called it will consider the given agent to be the computer and the gameState to be the current board. It will run through all of the possible successors of the current state for the current agent and run minimax on each successor, saving the one with the best evaluation and returning it. Our minimax also implements caching, saving the min and max evaluation for each visited state so that if a state is re-encountered, we can skip minimax and just return the evaluation.
- invalidCoordinate(coord, gameState, agent)
    - This simple utility function simply returns if a given coordinate(tuple) is a valid coordinate for the current agent and game state. It checks for bounds and overlaps with another piece that is different than the current agent. This is a useful subroutine for checking functions like getSuccessors and isValidMove when testing positions and orientatations for the L.
- getLegalDotPos(nextGameState, dot1, dot2)
    - This function generates all legal neutral piece positions for the given *next* game state and takes in the current positions of the neutral pieces as tuples. It returns a list of tuples of tuples (pairs of coordinates) where each element is a valid pair of neutral piece coordinates for the current L move. While this function has only one use, making it not really a subroutine, it helps with the readability of our getSuccessors function.
- getDotCoords(gameState)
    - This is a utility function to return the coordinates of the neutral pieces in a given state as a tuple of tuples(coordinates). This is useful for when you need to extract the neutral piece coordinates for situations like testing possible moves, the heuristic, etc. It is a valid subroutine.
- getCurrentLCoords(gameState, agent)

        - This is a utility function to return the L coordinates of a given agent in the given game state as a list of tuples(coordinates). This is useful for when you need to extract the L coordinates from a state to be used in many different places like the heuristic, checking validity of a move, etc. It is a valid subroutine.
    - compareLCoords(lCoords1, lCoords2)
        - This is a utility function to compare the L coordinates of two given L's where the L's are represented using a list of tuples that represent coordinates. It will compare and see if every coordinate in lCoords1 list matches the coordinates in lCoords2 list in any other. This is a useful subroutine for when we need to compare the L positions in different states.
    - getSuccessors(gameState, agent)
        - Each action is represented by a successor of the current game state. We chose to represent all valid orientations and positions as lists. By looking at the current game state, we iterated through the list of possible orientations and for each orientation, using a comparison function to check if the possible orientation is valid for the current orientation.
    - isValidMoveFormat(move)
        - This function accepts the string move that the user enters and simply checks if the string is in the right format of input (number number direction number number number number). This is a valid subroutine that is only used in the isValidMove function as preliminary step to verify if the format of the user's input create an actual move whether its legal or not.
    - coordOutOfBounds(coord)
        - Given a coordinate represented through a tuple (x,y), this function returns if the coordinate is out of bounds of the board or not. This function is a good utility function for other bigger "checking" functions that would need to validate coordinates.
    - isValidMove(gameState, agent, move)
        - This function takes in the current game state, the current playing agent, and the entered move from the user. It verifies that the move fits the valid format by calling isValidMoveFormat and then verifies that the move the user is trying to make works - there are no conflicting coordinates, overlapping Ls, if the moved dots have the right initial position. This function is a useful utility function because it allows us to separate checking if a user's move is valid from the rest of the game logic. It can exist as its own subroutine.
- Heuristic Explanation: Our heuristic has 7 components - moving factor of current agent, moving factor of opponent, central control factor of current agent, central control factor of opponent, a super component to handle near win/loss cases, an anticycle component, and a neutral piece proximity component. The moving factors of each agent are computed by computing the number of possible moves from the given next state and multiplying that by a factor, favoring more moves for the current agent and less for the

opponent. With respect to the move factor, this heuristic is offensive, favoring less moves for the opponent over more moves for the current agent. The central control factors are computed by computing the overall distance of each piece of the L to the center 4 pieces, where more pieces result in a higher value. These factors are neither weight defensively nor offensively. The super component comes into play only when there is a near win or loss - if the next move causes a loss, we make the super component a large negative number to override the other components in strong disfavor of the proposed successor state; if the next move causes a win, we make the super component a large positive number to override the other components in a strong favor of the proposed successor state. The anticycle component is calculated by seeing if the next state is already visited, since visited states indicate a cycle. If it is, we add a negative component, otherwise the component is position, favoring no cycles over a cycle. Finally, the dotProximity component favors staying away from the dots by calculating each coordinate in the L's proximity to the dot (manhattan distance) and adding them.

- Relevant Issues:
    - The maximum depth our game could run under a minute per move was a depth of 25. We increased the maximum depth in order to increase the branching factor while also changing our heuristic to maximize efficiency. To calculate the branching factor, we must figure out how many moves there are at any given state. There are 16 orientations for an L piece, of which only 8 are valid with an average of ~5. For each L, there are 8 empty spaces for the neutral pieces, giving us 8 x 7 = 56 possible pairs of neutral pieces per L. Our branching factor is #ofLMoves x #ofNeutralPieceMoves = ~334. There are around the same amount of total vs leaf nodes since at a depth of 25, leaf nodes dominate the node pool. This number is around 334^{25}, which is the average number of terminal states. Cycles can occur, and we can deal with them by keeping track of visited states and changing the weighting of our heuristic to not favor visited states over unvisited states.
    - We implemented a bonus feature that allows the user to ask for a hint during the game.