Varun Patro

105 296 500

Project 3 Report

Description of Design

In addition to the files given to me, I added a Support.h, which contains an interface and implementation of a Coord class. A Coord just contains a row and a column as private member variables, and the two public member functions return a row and a column respectively.

*Data Structures Used*

The major data structure I used was a stack. I used stacks to keep track of potential moves a player could make in SmartPlayer::chooseMove. I used them to keep track of moves made to the Scaffold so that I could easily undo by popping the stack. And I also used them extensively in the Game::completed() function, because I would push and pop Coord variables as I checked to see if consecutive checkers were the same color.

Something I feel worth mentioning is that the 2 by 2 vector I used to represent a scaffold was indexed at 0, while the actual game board was represented by level and columns ranging from 1 to the number of levels and columns respectively. It was therefore necessary to do some mapping in the checkerAt() function to convert the board coordinates into coordinates on my vector.

*Additional Functions*

For smartPlayer::chooseMove, I implemented a rate function that took in a Scaffold by reference, an integer number needed to win, two colors, a Boolean to keep track of whether I was maximizing or minimizing, and a depth parameter.

The rating function was used to rate the scaffolds. If the scaffold represented a win or tie, a specific rating was returned, and if the scaffold represented a game whose winner had not yet been decided, it iterated through the possible moves and recursed to get a rating when the game was eventually played out.

Description of Design for SmartPlayer::chooseMove

This function begins by making a copy of the scaffold passed to it. It looks at that scaffold and stores in a stack the columns that potentially can be moved into.

For each of these potential moves, it then calls the rate function (described later), which rates each scaffold by playing out all the different permutations that can result from it until an end result is reached. It stores this rating and its corresponding move in separate variables and then moves on to the next potential move. If this move results in a more advantageous scaffold (which will be determined by its rating which the rate function will return), then the variables storing the best rating and the best move will be updated with the new ones. After each potential move is evaluated, it is popped off the stack and undone from the copy of the scaffold.

chooseMove then returns the best move (which corresponds to the most advantageous rating).

Pseudocode:

Scaffold.cpp

checkerAt(…)

*return mapped location on vector of coordinates given as parameters (see Mapping section above)*

display()

*create a new vector with new dimensions (2\* # of columns + 1) columns and (# of levels + 1) rows*

*repeatedly:*

*if column is an even number*

*in bottom row, place ‘+’*

*else*

*place ‘-’*

*repeatedly:*

*repeatedly through columns in original vector:*

*if location is vacant*

*denote vacancy*

*if red*

*denote red*

*if black*

*denote black*

*repeatedly:*

*repeatedly through columns in enlarged vector:*

*print content of location*

*print new line*

makeMove(…)

*repeatedly (from bottom to top level):*

*if location is empty*

*set grid location to color parameter*

*push location onto stack*

*return true*

*return false*

undoMove(…)

*set up count variable*

*repeatedly:*

*repeatedly:*

*if location is vacant*

*increment count*

*if count is equal to area of vector*

*return 0*

*using coordinate at top of stack, set grid location of last move to vacant*

*save column of top coordinate of stack*

*pop stack*

*return column*

Player.cpp

HumanPlayer::chooseMove(…)

*Unitialized int*

*Repeatedly:*

*Prompt user for column input*

*Save input*

*If valid input*

*If input location is vacant*

*Return column*

*Return 0*

BadPlayer::chooseMove(…)

*Repeatedly:*

*Repeatedly:*

*If location is vacant*

*Return column*

*Return 0*

SmartPlayer::chooseMove(…)

*Make a copy of the scaffold*

*Set new variable to opponent of color parameter*

*Repeatedly (through columns):*

*If available*

*Store it*

*Create variables for best rating and best move*

*Repeatedly (through all valid moves):*

*Make move on the copied scaffold*

*Call rate on updated scaffold*

*If returned rating is greater than variable*

*Set variable to returned rating*

*Set move*

*Pop the move*

*Undo the move*

*Return the best move*

Rate(…)

*Create variable to opponent of toPlay parameter*

*If game complete (call function):*

*If tie*

*Return tie rating*

*If loss*

*Return loss rating (account for depth)*

*If win*

*Return win rating (account for depth)*

*Else*

*Create copy of scaffold*

*Create variable for best rating*

*Repeatedly (through all columns):*

*If column is viable*

*Store it*

*Repeatedly (through all possible moves):*

*Make the move on copy*

*Call rate recursively on it*

*If maximizing*

*If rating (returned by rate) greater than variable*

*Set variable equal to return value of rate*

*Else (if minimizing)*

*If rating (returned by rate) less than variable*

*Set variable equal to return value of rate*

*Undo the move*

*Pop the move*

*Return the best move*

Game.cpp

*Completed(…)*

*If number of empty spaces equals area*

*Return false*

*Create copy of scaffold*

*Save move of last column in copy (undo copy’s last move)*

*Create variables for row, column and color of last move*

*Create stack of Coords*

*Repeatedly through levels in scaffold, NOT copy:*

*If spot at last column and loop level is not vacant, is the location of the last move*

*Set last color*

*Set column*

*Set row*

*Break*

*Create variables and store row and column*

*Push coordinate of last move onto stack*

*Create counter*

*Repeatedly (while stack is not empty):*

*Pop stack*

*If in bounds and checker is color of last move*

*Push location onto stack*

*Decrease one row*

*Increment count*

*If N checkers in a row found*

*Set winner to last color*

*Return true*

*Push saved last coordinate onto stack*

*Reset counter*

*Reset column and row*

*Repeatedly:*

*If column in bounds and location same color*

*Push location onto stack*

*Move column to the right*

*Increment count*

*If N in a row*

*Set winner*

*Return true*

*Push last onto stack*

*Move column to left from original*

*Repeatedly:*

*If column in bounds and checker same color*

*Push location onto stack*

*Move column left*

*Increment counter*

*If N in a row*

*Set winner*

*Return true*

*Repeat for both diagonal directions*

takeTurn()

*if game complete*

*return false*

*if game board empty*

*make red play*

*else*

*make a copy of scaffold*

*undo move to figure out who went last*

*if red*

*make black play*

*else*

*make red play*

*return true*

play()

*repeatedly (while game not complete):*

*make someone play*

*display the board*

*make user enter to display moves in a reasonable time scale*

Difficulties

My biggest challenge in this program was implementing the minimax algorithm. My big problem here was to figure out whether I was minimizing or maximizing. In a more general sense though, it was difficult to think about this recursive algorithm because there were so many factors to consider initially (keeping track of the color, keeping track of depth, keeping track of moves and ratings). And after understanding the algorithm, putting it into C++ was an even bigger challenge, especially trying to adapt it to the specifics of this particular game. This one part of the project took me several days.