Shaan Mathur

Smallberg

CS32

May 20, 2016

Project 3 Report

Class Design

ScaffoldImpl

class ScaffoldImpl

{

private:

vector<int>\* col;

stack<int>\* moves;

int numLvl, numCol;

int vacancies;

public:

ScaffoldImpl(int nColumns, int nLevels);

~ScaffoldImpl();

ScaffoldImpl(const ScaffoldImpl& s);

int cols() const;

int levels() const;

int numberEmpty() const;

int checkerAt(int column, int level) const;

void display() const;

bool makeMove(int column, int color);

int undoMove();

//int mostRecentMove() const;

ScaffoldImpl& operator=(const ScaffoldImpl& s);

void swap(ScaffoldImpl& s);

};

My ScaffoldImpl maintains a record of the checkers by having a dynamically allocated array of vectors of ints, and records all moves by using a stack of ints. This decision was made because the number of columns are going to be some constant, unchanging value throughout the use of the Scaffold, but the number of levels can be implemented in a different manner. The use of column vectors are as follows: the vector only holds values if they are RED or BLACK. It does not hold VACANT values. Thus, numLvl – col[i].size() == number of VACANTs there should be in the column. This at first doesn’t seem to give any value (and perhaps could be an implementation headache), but it proved useful when needing to quickly undo a move. I would use the most recent move (that was pushed onto the stack for every move) and then I would take that column value, go to the respective column, and pop\_back(). It does not require any looping through values until the last RED or BLACK is found. It is quick and nicely efficient.

GameImpl

class GameImpl

{

private:

Scaffold scaf;

int n;

Player\* red, \*black;

bool redTurn, gameOver;

bool examine(int color) const;

bool opt\_examine(int color) const;

int col\_new;

public:

GameImpl(int nColumns, int nLevels, int N, Player\* red, Player\* black);

bool completed(int& winner) const;

bool takeTurn();

void play();

int checkerAt(int c, int r) const;

};

No major data structures were necessary for GameImpl. It’s primary focus was the maintenance of a Scaffold and two Players in the context of the game. I did define several extra helper functions. bool examine(int color) const , for example, quickly parsed through the game to see if that color won. Yes, it would be more efficient to only look at the most recent move to see if it caused a win. But this algorithm in my experimentation has been remarkably quick, and I did have more personal trust in it than my other algorithm bool opt\_examine(int color) const. That algorithm did exactly the other method I just described, of examining the most recent move and checking around it to see if it instigated a win. I will describe my examine() methods in better detail soon.

PlayerImpl

class HumanPlayerImpl

{

public:

int chooseMove(const Scaffold& s, int N, int color);

};

class BadPlayerImpl

{

public:

int chooseMove(const Scaffold& s, int N, int color);

};

HumanPlayerImpl was very straightforward implementation. Some prompts, a couple of checks, a cin, checking if that was a valid value to continue forward, and finally returning that value. It is nothing special from what we’ve done in CS31.

BadPlayerImpl was also fairly trivial. A single for loop went through all the columns and checked the top level at each specific column to see if it was VACANT. If so, it would put a checker there. If it went through the loop and it turned out the Scaffold was full, it would return

-1. Thus, it would choose the leftmost unfilled column to submit its chip into.

class SmartPlayerImpl

{

public:

int chooseMove(const Scaffold& s, int N, int color);

};

int opt\_examine(const Scaffold& scaf, int n, int col\_new, int color, int depth);

int standard\_examine(const Scaffold& s, int N, int color, int depth);

int rate(Scaffold& s, int N, int color, int depth, int col\_new);

SmartPlayerImpl featured the most interesting algorithm. I created three helper functions (it seems extreme but it was useful): opt\_examine(…), examine(…), and rate(…). It used a modified (for the better) version of the minimax algorithm that figured out what move would yield the best results according to the specs’ rules.

First, the SmartPlayerImpl::chooseMove(…) wanted to figure out which column would be the best one to move a checker into in regards to the quickest win (or prolonged loss). It thus traversed every column, inserted a checker into that column, and asked the same question to the rate(…) function (which is the same thing. chooseMove is an “auxiliary” function as Professor Smallberg used the term in class).

Rate(…) would ask itself the same question, alternating personas depending on the depth of recursive call. The first call to rate(…) would take the persona as a Human, while the next one would be SmartPlayer. If there have been 2\*N – 1 moves (in a Connect-N game), then now it is possible for a particular Scaffold to have a winning combination. My standard\_examine(…) traversed the entire Scaffold (in a fairly optimized manner, ignoring duplicates and the such) and returned a 0, -1\*(numberEmpty + 1), or 1\*(numberEmpty + 1). A positive is a Computer win, negative a human win, 0 a potential tie (which can be determined if you ask the Scaffold if there are no more empty spaces). I multiply the 1 and -1 against (numberEmpty + 1) because then my logic will later be right. For now, just know that quick wins will be higher in value, and later losses will be higher relative to quick losses. For all iterations deeper than 2\*N – 1, I use opt\_examine(…) which only considers the most recent move to see if it resulted in a win. I found empirically that using both standard\_examine(…) AND opt\_examine(…) in this manner yielded the quickest results.

Rate(…) finally returns a positive (higher for quicker wins), a negative (prolonged loss being -1), or a 0 for a tie. Using this, every persona decides which one it wants, and returns that. Humans want very low negatives, and Computers want the highest positive. Finally the chooseMove(…) gets all the data, gets the maximum, and returns that value upward to whomever invoked the function.

Pseudocode

chooseMove(…)

Declare a bestIndex and assign it 1

Declare a bestCaseValue and give it a very negative number

For every column

If that column is not VACANT at the top

Continue

Insert a checker into that column

Assign to temp rate(current state of the Scaffold)

If temp is greater than bestCaseValue

bestIndex is assigned current column

bestCaseValue is assigned temp

undo the last move

rate(…)

if the number of moves is than 2N-1

Invoke standard\_examine() to see if there is a win/loss/tie

If a win or loss

Return that value

If the scaffold is empty

Return 0

Else if the number of moves is greater than 2N-1

Invoke opt\_examine() to see if there is a win/loss/tie

If a win or loss

Return that value

If the scaffold is empty

Return 0

Declare variables for worst and best values

For every column

If that column is not VACANT at the top

Continue

Insert a checker into that column

Assign to temp rate(current state of the Scaffold)

If temp is greater than best

best is assigned current column

If temp is less than worst

Worst is assigned current column

undo the last move

If depth is odd

Return worst

Else

Return best

**Completed(…) uses my examine(…), so I’ll write the pseudocode for that instead.**

Examine(…)

For all levels

For all columns

If current checker isn’t the color specified

Continue

If the checker to the right is of the same color

Keep counting in that direction

If that count hits N, return true

If the checker to the bottom is of the same color

Keep counting in that direction

If that count hits N, return true

If the checker diagonally downwards to the right is of the same color

Keep counting in that direction

If that count hits N, return true

If the checker diagonally downwards to the left is of the same color

Keep counting in that direction

If that count hits N, return true

Return false

Bugs

One problem about SmartPlayer::chooseMove(…) is that it assumes all opponents are intelligent. Because the algorithm is recursive and it in itself is intelligent, when it takes on the persona of a Human, it assumes that human is intelligent as well. All decisions are dependent on this intelligence, and it can pose an issue. What if an extremely dumb player did something that was not anticipated to happen? Then the code may fail.

Additionally, if chooseMove(…) finds that all columns are equally bad, it will choose the leftmost one arbitrarily. However, this is not good! What if one pathway led to more ties than the other one? One (-1) pathway might lead to possible (-1, 1) or another would lead to (-1, 0,0,0,0,0,1). But because of the nature of the algorithm, the other values are dropped and we only see the -1. If there could be a way around this, it would be extremely beneficial to the AI.

Test Cases

/\*

Scaffold s(5, 3);

int col\_new = 0;

s.display();

int count = 0;

//This runs a demo of the Scaffold class

do {

cin >> col\_new;

if (count % 2 == 0)

{

s.makeMove(col\_new, BLACK);

s.display();

if (t\_opt\_examine(s, col\_new, 3, BLACK))

break;

}

else

{

s.makeMove(col\_new, RED);

s.display();

if (t\_opt\_examine(s, col\_new, 3, RED))

break;

}

count++;

} while (true);

\*/

/\*const int N = 3;

const int COLS = 3;

const int LVLS = 4;

Scaffold test(COLS, LVLS);

int x = 0;

test.display();

//This runs a test AI algorithm against the makeMove, to see if the algorithm

//provides valid indices

while (test.numberEmpty() != 0) {

cin >> x;

test.makeMove(x, RED);

test.makeMove(t\_chooseMove(test, N, BLACK), BLACK);

test.display();

}

\*/

//This tests another component of the AI algorithm while in tandem also testing //the Scaffold class

/\*

const int CONNECT\_N = 5;

Scaffold test(10, 10);

int x = 0;

cin >> x;

int count = 0;

while (x != -1) {

if (count % 2 == 0)

test.makeMove(x, RED);

else

test.makeMove(x, BLACK);

test.display();

count++;

if (count >= 2 \* CONNECT\_N - 1) {

cout << t\_examine(test, CONNECT\_N, RED, 1) << ", " << t\_examine(test, CONNECT\_N, BLACK, 1) << endl;

}

cin >> x;

}

\*/

/\*

//These are a rigorous amount of test cases for all functions in Scaffold

Scaffold t(100, 100);

int count = 0;

for (int i = 1; i <= 100; i++) {

for (int j = 1; j <= 100; j++) {

if (count % 2 == 0)

t.makeMove(j, RED);

else

t.makeMove(j, BLACK);

count++;

}

}

fout << "Done filling" << endl;

for (int i = 2; i <= 100; i++) {

if (i == 100)

cerr << "Surprise " << endl;

fout << "Working on Connect-" << i << endl;

fout << t\_examine(t, i, RED, 1) << ", " << t\_examine(t, i, BLACK, 2) << endl;

}

//t.display();

fout.close();

cout << t\_examine(t, 1, RED, 1) << ", " << t\_examine(t, 1, BLACK, 2) << endl;

\*/

/\*

Scaffold a(2,3);

cout << "Columns:\t" << a.cols() << endl;;

cout << "Levels:\t" << a.levels() << endl;;

cout << "Number Empty:\t" << a.numberEmpty() << endl;

a.display();

assert(a.makeMove(1, BLACK));

a.display();

assert(a.makeMove(1, RED));

a.display();

assert(a.makeMove(1, BLACK));

a.display();

assert(!a.makeMove(1, BLACK));

a.display();

assert(a.makeMove(2, BLACK));

a.display();

assert(a.makeMove(2, RED));

a.display();

assert(!a.makeMove(2, VACANT));

assert(a.makeMove(2, BLACK));

a.display();

assert(!a.makeMove(2, RED));

a.display();

assert(!a.makeMove(3, RED));

for (int i = 0; i < 100; i++) {

assert(!a.makeMove((i % 2) + 1, RED));

}

for (int i = 3; i < 50; i++) {

assert(!a.makeMove(i, BLACK));

}

Scaffold b(2, 3);

assert(b.checkerAt(1, 3) == VACANT && b.checkerAt(1, 2) == VACANT && b.checkerAt(1, 1) == VACANT);

assert(b.checkerAt(2, 3) == VACANT && b.checkerAt(2, 2) == VACANT && b.checkerAt(2, 1) == VACANT);

assert(b.makeMove(1, RED));

b.display();

assert(b.checkerAt(1, 1) == RED);

assert(b.makeMove(1, BLACK));

b.display();

assert(b.checkerAt(1, 2) == BLACK && b.checkerAt(1, 1) == RED);

assert(b.makeMove(1, RED));

b.display();

assert(b.checkerAt(1, 3) == RED && b.checkerAt(1, 2) == BLACK && b.checkerAt(1, 1) == RED);

assert(b.makeMove(2, BLACK));

b.display();

assert(b.checkerAt(1, 3) == RED && b.checkerAt(1, 2) == BLACK && b.checkerAt(1, 1) == RED);

assert(b.checkerAt(2, 1) == BLACK);

assert(b.makeMove(2, RED));

b.display();

assert(b.checkerAt(1, 3) == RED && b.checkerAt(1, 2) == BLACK && b.checkerAt(1, 1) == RED);

assert(b.checkerAt(2, 2) == RED && b.checkerAt(2, 1) == BLACK);

assert(b.makeMove(2, BLACK));

b.display();

assert(b.checkerAt(1, 3) == RED && b.checkerAt(1, 2) == BLACK && b.checkerAt(1, 1) == RED);

assert(b.checkerAt(2, 3) == BLACK && b.checkerAt(2, 2) == RED && b.checkerAt(2, 1) == BLACK);

int t = 100;

while ((t = b.undoMove()) != 0) {

b.display();

cout << t << endl;

}

cout << t << endl;

assert(b.checkerAt(1, 3) == VACANT && b.checkerAt(1, 2) == VACANT && b.checkerAt(1, 1) == VACANT);

assert(b.checkerAt(2, 3) == VACANT && b.checkerAt(2, 2) == VACANT && b.checkerAt(2, 1) == VACANT);

{

Scaffold c(a);

assert(!c.makeMove(3, RED));

for (int i = 0; i < 100; i++) {

assert(!c.makeMove((i % 2) + 1, RED));

}

for (int i = 3; i < 50; i++) {

assert(!c.makeMove(i, BLACK));

}

}

assert(!a.makeMove(3, RED));

for (int i = 0; i < 100; i++) {

assert(!a.makeMove((i % 2) + 1, RED));

}

for (int i = 3; i < 50; i++) {

assert(!a.makeMove(i, BLACK));

}

\*/