University of the Witwatersrand

COMS3005: Advanced Analysis of Algorithms

Peg Solitaire Backtracking Assignment

October 31, 2017

*B*y Bancroft, E. (879192)
*A*nd Chalom, J. (711985)

1 Introduction

The purpose of the assignment is to implement and analyse a version of the Peg Solitaire game and the backtracking algorithm (to play the game to completion and return a valid path).

2 Background

Peg Solitaire is a board game which has a number of holes that can be filled with pegs. We have chosen to use the European/French style of board which has four extra positions for pegs on the board 2. In this style most of a grid of 49 has peg holes excluding three per corner resulting in 37 peg holes. The aim of the game is to remove pegs until only one peg remains. This is the position one row directly above the central peg, a row above that and the left most peg in the top row. Moves are made when pegs jump over a peg and are placed in an open position. Then a peg which has been jumped over, is then removed. This move can happen in both horizontal and vertical directions. In the European variant, the game has three possible optimal terminal states. Its is possible to hit sub-optimal states where there are more pegs left on the board but no possible moves left [1].

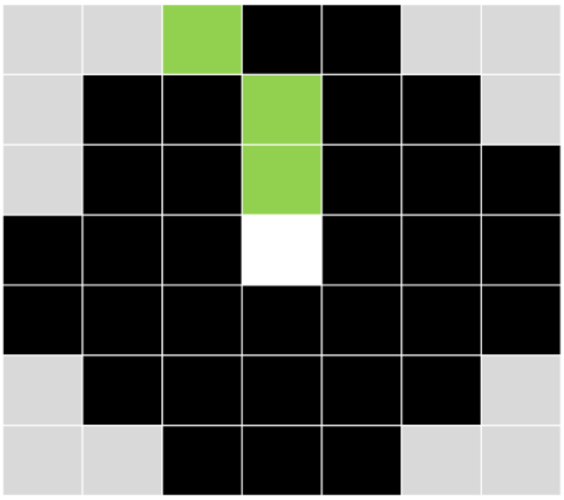

Figure 1: Diagram of an European Peg Solitaire Board Where the Black Squares Represent Peg Positions, Green Are Terminal Positions, Grey Are Not Positions and White is the Central Pixel.

The backtracking algorithm is similar to a brute force approach to finding solutions to problems but is more systematic. It attempts to follow a logical series of decisions in solving these problems and when a block state occurs the algorithm will backtrack to previous decisions and choose different paths until a terminal (complete) state is reached. The full set of solutions to a problem can be found by continuing to run the algorithm until all paths have been searched but that is not always necessary.

Algorithm 1 Recursive Algorithm (From References [2])

```
procedure FindSolution(start, final, path)
    if start.numPegs <= final.numPegs then
        return (start = final)
    else
        for each jump J ∈ [0,n) x [0,m) x {NORTH,EAST,SOUTH,WEST} do
            if J is a legal jump for start then
                start.makeMove(J)
                path.push(J)
                found = FindSolution(start, final, path)
                if found then
                    return TRUE
                else
                    start.makeReverseMove(J)
                    path.pop()
        return FALSE
```

2.1 Stack Based Algorithm

The stack based algorithm we used is adapted from the recursive version. The purpose of implementing the stack based algorithm was to compare it's effciency to that of the recursive based algorithm. Also the recursive algorithm was expected to be much slower.

Algorithm 2 Stack Based Algorithm (Adapted From References [2])

```
procedure FindSolution(start, outPath, totalNumPegs, numValidMoves)
    currentState ← start
    path ← start.getMoves()
    numPegs ← currentState.getNumPegs()
    found ← FALSE
    i ← 1
    stackVector.push(path)
    boardVector.push(currentState)
    while found is FALSE and i <= numPegs and stackVector.size() > 0 and path.size() > 0
do
        path ← stackVector.pop()
        currentState ← boardVector.pop()
        while currentState.checkGameEnd() != FALSE do
            numPegs ← currentState.getNumPegs()
            move ← path.pop()
            if currentState.checkIfMoveValid(move) == TRUE then
                stackVector.push(path)
                boardVector.push(currentState)
                numValidMoves = numValidMoves + 1
                currentState.makeMove(move)
                outPath.push(move)
                path ← currentState.getMoves()
                numPegs ← currentState.getNumPegs()
        numPegs ← currentState.getNumPegs()
        if currentState.checkGameWin() == TRUE then
            found = TRUE
        else
            found = FALSE
    return currentState
```

3 Implementation

3.1 Technology Used

We made use of c++ 11, its standard libraries and OpenMP to time our results. Our results are saved as comma seperated value files which are then processed and graphed by libre office.
To create the stacks in the stack implemntation we used vectors.
To generate the pseudorandom numbers we used the Mersenne Twister engine that comes included in the c++ "random" library.

3.2 How To Compile and Run

In order to compile and run the code:
Go to root folder of the project and run make.
Then run ./bin/game.out to run the game.

Commandline Parameters:

```
Usage Example: ./bin/game.out -rb
Random state: ./bin/game.out -rr
Full state: ./bin/game.out -rf
Run Stacked Based Backtracking: -rb
Run Recursive Backtracking: -recurse
Manual: -m
Help: -h
```

3.3 Our Termination Conditions

Although both algorithms terminate under similar theoretical conditions (no more moves possible or it has reached a "win" state), due to their different implemntations, the teminating conditions are diffrent.

3.3.1 Recursive Implementation

This implentation works very similar to a Depth First Search and will terminate under very similar conditions to a DFS algoritm.
These conditions are :

- Found one of the three "win" states.
- Cannot backtrack any further: So the algorithm has returned to the root layer, and as such hasn't found any of the "win" states, and wishes to try backtrack further, however there are no more layers to backtrack too and as such the algorithm will conclude that a "win" state cannot be found given the current start state and will terminate.

3.3.2 Stack Implementation

This implementation works diffrently from the recursive as it uses a stack of paths to store the progress down a branch of a tree and create backtrack points when ever the algorithm does a move.
The conditions where the stack implemntation will terminate are :

- Found one of the three "win" states.
- If the algorithm finds it's self on the orginal path that was generated and the path size of that path is zero (poped off all moves) . i.e. There is nowhere higher to back track too, and all moves on the current path have been exasusted then the algorithm has tried all possible moves and will conclude that a "win" state cannot be found given the current start state and will terminate.

3.4 Best Case

We decided that it would be helpfull to have best case data points for both recursive and the stack implemntation to compare with. These best case game states always incude a single path that will result in finding a "win" state.
However there is a major issue that we ran into regarding the best case state generator. It only worked up to 17 pegs. This due to the way we generate the backwards path. i.e it will start at a "win" state and try a up reversal, if it can't, it will try a right move reversal,if it can't, then try a down reversal and if that fails try a left move reversal.
Contiueing this pattern the board will be filled with n pegs if n<=17. When we try creating 18+ points it will fail to find any valid reverse move and as such 17 points is the maximum number of points we could create with this algorithm.

3.5 Problems We Encountered

One of the most notable feature/Problem we encountered was that the stack implemntation was incrediably fast relative to the recursive implemntation. Both returned the correct output but because of the recursive implemntation relying on a heap, we only managed to get 17 data points in 30 hours of the program running.
The stack implemntation doesn't use the heap (which the recursive alogrithm does), which is internally slow, so therefore allowing the stack implemntation's effciency to far surpass that of the recursive's.
The vector stack in c++ also is a much more effcient use of memory compared to storing objects on a heap.

3.6 Experimental Setup

For our experiment we tested with an increasing number of pegs being place randomly throughout the board.
Each test will try to find one of the three "win" states based on the starting state that was generated. This proccess will be timed and the time, together with the number of pegs that were generated will be recoreded.
Although with the recursive implemntation we have to run the same state three times (once for each of the three possible "win" states) the time that is being recorded is the amount of time that it takes to run the instance of the algoritm that finds a "win" state, or the time it takes for one instance to explore the entire tree, not all three.

4 Theoretical Analysis

We assume that our basic opertion used for analysis in the backtracking algorithm for peg-solitaire is generating a new state which is playing a valid move or jumping a peg into a valid empty space (and removing a peg between them). Determining if there are no more moves, and also getting every valid move for a peg are both assumed to take constant time, which is the speed of each conditional by the number of board elements i.e. 49 elements.

The best case complexity of backtracking for peg-solitaire would be the case where only one path needs to be generated for any number of pegs i.e. no backtracks occur because a game win is found at the end of the first path. In this case the complexity is the number of pegs left on the board or the length of the found path. If we assume this number is represented by the variable n, then the best case complexity is $O(n)$.

In the worst case complexity event no game win is possible so every possible path has to be traversed by the algorithm. Each peg has the potential to move in four directions but that is unlikely. From our observations of the game being played out moves on a board from a start state tends to allow on average two possible moves per peg when moves are available. This means that on average (based on our observations) the game has a branching factor of 2. Since each path can be considered to be a branch on a tree data structure and the number of branches is the number of pegs which is assumed to be n. This means that the total search space (game-space) size is 2^n and so the worst case complexity is directly propotional to this number i.e. the worst case complexity is $O(2^n)$.

5 Results

5.1 Graphs

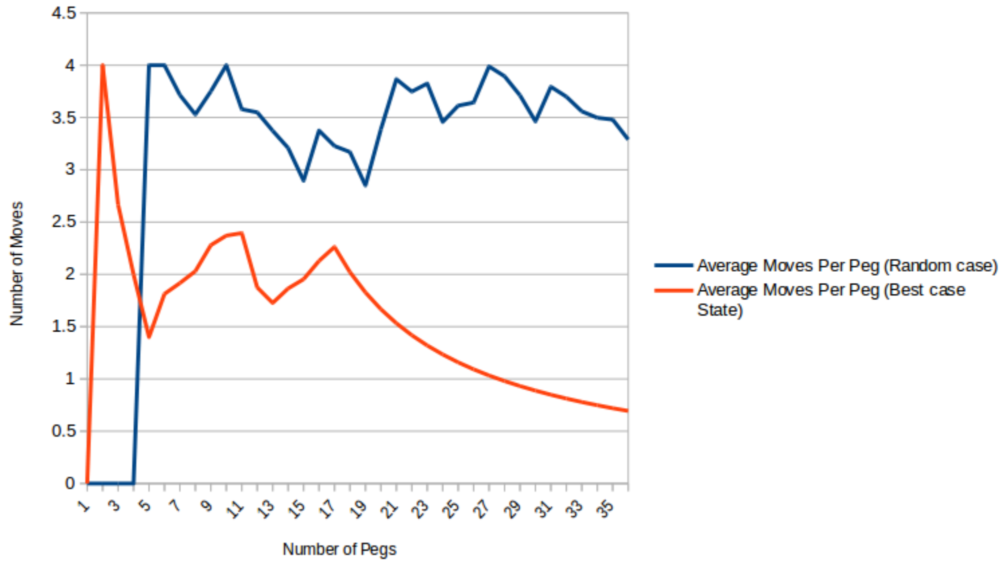

Figure 2: Average Available Moves Per Peg at Each Iteration of Stack Based Algorithm

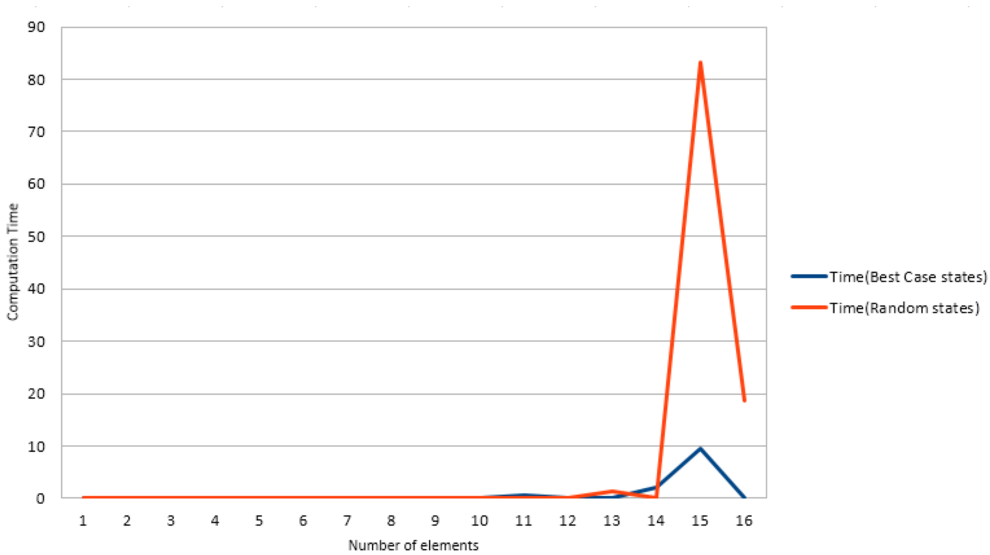

Figure 3: Timed Results of Number of Pegs Versus Completion Time for Recursive Implementation

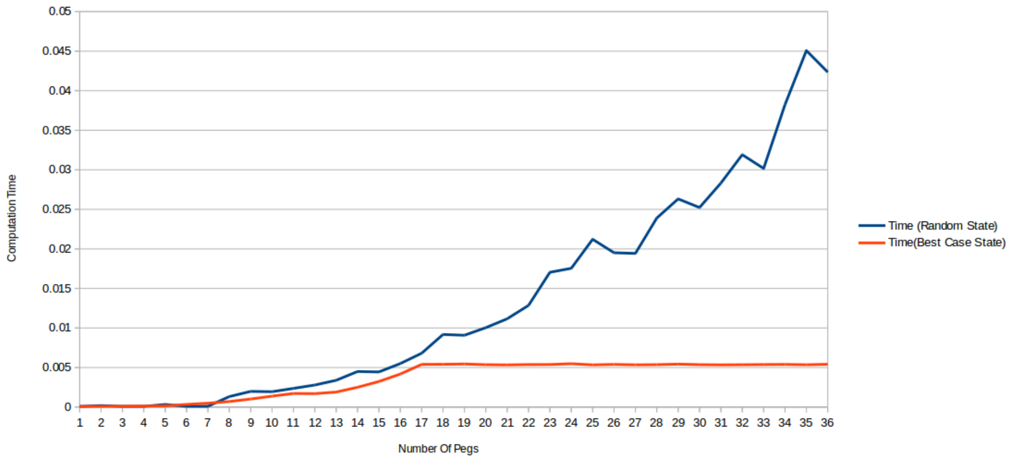

Figure 4: Timed Results of Number of Pegs Versus Completion Time for Stack Implementation

Appendix B contains the tables of generated results.

6 Empirical Analysis

7 Conclusion

8 Group Member Contribution

Member	Evan Bancroft 879192	Jason Chalom 711985
Game	75%	25%
Back Tracking Algorithm	50%	50%
Report	25%	75%

Table 1: Contributions of Group Members By Task

Acknowledgements

All drawn diagrams were drawn using http://draw.io/ and charts were made with Libre Office. All the programming was done in c++ using OpenMP for its timing functions.

References

[1] Douglas Wilhelm Harder. Peg solitaire. https://ece.uwaterloo.ca/~dwharder/aads/Algorithms/Backtracking/Peg_solitaire/.

[2] Charles E. Leiserson. Lab 5: Backtracking search. http://courses.csail.mit.edu/6.884/spring10/labs/lab5.pdf, 2010.

Appendix A: Source Code

8.1 Source: backtracking.cpp

```
GameBoard backtracking_stack(GameBoard start, vector<vector<int>> &outPath, int &
    totalNumPegs, int &numValidMoves)
{
    bool found = false;
    GameBoard current;
    std::vector<Move> path;
    start.getMoves(path);
    current.copy(start);

    std::vector<std::vector<Move>> stackVector;
    std::vector<GameBoard> boardVector;
    stackVector.push_back(path);
    boardVector.push_back(current);

    int numPegs = current.numPegs();
    int i = 1;

    while (found == 0 && i <= numPegs && stackVector.size() > 0 && path.size() > 0)
    {
        path = stackVector.back();
        stackVector.pop_back();
        current.copy(boardVector.back());
        boardVector.pop_back();

        //print stuff
        current.printBoard();
        cout << "back tracked path \t V:" << stackVector.size() << "\tP:" << path.size() <<
            endl;

        while (!current.checkGameEnd() && numPegs >= 1 && path.size() > 0) // do the path
            till no path left
        {
            numPegs = current.numPegs();
            totalNumPegs += 1;

            Move mov = path.back();
            path.pop_back();

            if (current.checkIfMoveValid(mov.id, mov.r, mov.c))
            {
                stackVector.push_back(path);
                boardVector.push_back(current);

                numValidMoves++;

                current.makeMove(mov.id, mov.r, mov.c);
                std::vector<int> coord;
                coord.push_back(mov.r);
                coord.push_back(mov.c);
                coord.push_back(mov.id);
                outPath.push_back(coord);

                current.getMoves(path);
                numPegs = current.numPegs();
```

```
        }
    }

    numPegs = current.numPegs();
    if (current.checkGameWin())
    {
        found = 1;
        // break;
    } else
    {
        found = 0;
        // break;
    }
  }

  return current;
}

bool backtracking_recursive(GameBoard start, GameBoard final, vector<Move> path)
{
  if (start.numPegs() <= final.numPegs())
  {
    return start.equals( final );
  }
  else
  {
    std::vector<Move> moves;
    start.getMoves(moves);
    for (Move J : moves)
    {
      if (start.checkIfMoveValid(J.id, J.r, J.c))
      {
        start.makeMove(J.id, J.r, J.c);
        path.push_back(J);

        bool found = backtracking_recursive(start, final, path);
        if (found)
          return true;
        else
          start.makeReverseMove(J.id, J.r, J.c);
        path.pop_back();
      }
    }
    return false;
  }
}

GameBoard bestCase(int numPegs)
{
  GameBoard bc;
  bc.board[0][2] = 1;
  //bc.printBoard();
  int curR = 0;
  int curC = 2;

  for (int i = 1; i < numPegs; ++i)// for each new peg
  {
    int j = 0;
```

```
bool found = false;
while (j < 4 && found == false) // try reverse up, right, down, left
{
    switch (j)
    {
    case 0: {//reverseUp
        if (curR < 5)
        {
            curR = curR + 2;
            if (bc.checkIfMoveValidReverse(j, curR, curC))
            {
                bc.makeReverseMove(j, curR, curC);
                found = true;
            } else
            {
                curR = curR - 2;
            }
        }
        break;
    }
    case 1: {//reverseright
        if (curC > 1)
        {
            curC = curC - 2;
            if (bc.checkIfMoveValidReverse(j, curR, curC))
            {
                bc.makeReverseMove(j, curR, curC);
                found = true;
            } else
            {
                curC = curC + 2;
            }
        }
        break;
    }
    case 2: {//reversedown
        if (curR > 1)
        {
            curR = curR - 2;
            if (bc.checkIfMoveValidReverse(j, curR, curC))
            {
                bc.makeReverseMove(j, curR, curC);
                found = true;
            } else
            {
                curR = curR + 2;
            }
        }
        break;
    }
    case 3: {// reverseleft
        if (curC < 5)
        {
            curC = curC + 2;
            if (bc.checkIfMoveValidReverse(j, curR, curC))
            {
                bc.makeReverseMove(j, curR, curC);
                found = true;
```

```
                } else
                {
                    curC = curC - 2;
                }
            }
            break;
        }
        default: {
            std::cout << "Invalid direcion \n";
            return false;
            break;
        }
        }
        j++;
    }
  }
  return (bc);
}
```

8.2 Source: solitaire$_board.h$

```
/*Class header for the game definitions and rules*/
/*Using the European style board layout
  The European style has no solution if the centre is empty however there are 3 postions
       where a valid solution is viable
  I will be only using 1 of those positions*/
/*Max 36 pegs in play at a time with 1 missing*/
/* Board's dimensions in a square are 7x7*/
/*12 positions are missing fromthe square, 3 in each corner in a triangle*/
/*Neeed to randomize the peg postions on the board*/
/* -1 repesents out of bounds area (off board),
   0 repesents empty space
   1 repesents a position with a peg
*/
#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#include <time.h>
#include <string>
#include <random>
#include "omp.h"

using namespace std;

class GameBoard
{
private:
   int row = 7;
   int col = 7;

public:
   /*varibles*/
   std::vector<std::vector<int>> board;

   /*constructors*/
   GameBoard();
   GameBoard(int num_pegs);
```

```
    /*Functions*/
    int getRow();
    int getCol();
    void setRow(int r);
    void setCol(int c);

    void printBoard();
    int numPegs();
    int numMoves();
    std::vector<std::vector<int>> getPegs();

    void euroConfig_Start();
    void euroConfig_Random();
    bool makeMove(int id, int r, int c);
    bool checkIfMoveValid(int id, int r, int c);
    bool checkIfMoveValidReverse(int id, int r, int c);

    void copy(GameBoard gb);
    bool equals(GameBoard gb);
    void getMoves(std::vector<Move> &path);
    bool checkGameEnd();
    bool checkGameWin();

    bool makeReverseMove(int id, int r, int c);

};

GameBoard::GameBoard()// default empty board exept the corners.
//37 0's and 12 (-1)'s
{
    setRow(row);
    setCol(col);

    //init rows of board
    for (int i = 0; i < getRow(); ++i)
    {
        std::vector<int> row;
        for (int j = 0; j < getCol(); ++j)
        {
            if ((i == 0) || (i == 6))
            {
                if ((j >= 2) && (j <= 4))
                {
                    row.push_back(0);
                } else
                {
                    row.push_back(-1);
                }
            } else if ((i == 1) || (i == 5))
            {
                if ((j >= 1) && (j <= 5))
                {
                    row.push_back(0);
                } else
                {
                    row.push_back(-1);
                }
```

```
            } else
            {
                row.push_back(0);
            }
        }
        board.push_back(row);
    }

}

GameBoard::GameBoard(int num_pegs)//default empty board populated with num_pegs of
    valid pegs at random positions.
//num_pegs 1's , 12(-1)'s and (37-num_pegs) 0's
{
    setRow(row);
    setCol(col);

    //init rows of board
    for (int i = 0; i < getRow(); ++i)
    {
        std::vector<int> row;
        for (int j = 0; j < getCol(); ++j)
        {
            if ((i == 0) || (i == 6))
            {
                if ((j >= 2) && (j <= 4))
                {
                    row.push_back(0);
                } else
                {
                    row.push_back(-1);
                }
            } else if ((i == 1) || (i == 5))
            {
                if ((j >= 1) && (j <= 5))
                {
                    row.push_back(0);
                } else
                {
                    row.push_back(-1);
                }
            } else
            {
                row.push_back(0);
            }
        }
        board.push_back(row);
    }

    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution<int> row_distribution(0, row - 1);

    std::random_device rd2;
    std::mt19937 mt2(rd2());
    std::uniform_int_distribution<int> col_distribution(0, col - 1);
    for (int i = 0; i < num_pegs; i++)
    {
```

```
        int rnd_row = row_distribution(mt);
        int rnd_col = col_distribution(mt2);
        while (!board[rnd_row][rnd_col] == 0)
        {
            rnd_row = row_distribution(mt);
            rnd_col = col_distribution(mt2);
        }
        board[rnd_row][rnd_col] = 1;
    }
}

int GameBoard::getRow()
{
    return row;
}
int GameBoard::getCol()
{
    return col;
}
void GameBoard::setRow(int r)
{
    row = r;
}
void GameBoard::setCol(int c)
{
    col = c;
}

void GameBoard::printBoard()//Prints the current board to console.
{
    cout << '\n';
    for (int i = 0; i < (int) board.size(); ++i)
    {
        for (int j = 0; j < (int) board[i].size(); ++j)
        {
            cout << board[i][j] << '\t';
        }
        cout << '\n';
    }
    cout << '\n';
}

int GameBoard::numPegs()//Number of valid pegs on the board
{
    int count = 0;
    for (int i = 0; i < (int) board.size(); ++i)
    {
        for (int j = 0; j < (int) board[i].size(); ++j)
        {
            if (board[i][j] == 1)
            {
                count = count + 1;
            }
        }
    }
    return count;
}
```

```
int GameBoard::numMoves()//Total possible number of moves
{
    int numMoves = row * col * numPegs();
    return numMoves;
}

std::vector<std::vector<int>> GameBoard::getPegs()//Retrieves the coordinates of the pegs.
      pegs[0][0]=r and pegs[0][1]=c of peg 0;
                                    //pegs[0][2] is direction id
{
    std::vector<std::vector<int>> pegs;
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            if (board[i][j] == 1)
            {
                std::vector<int> coord;
                coord.push_back(i);
                coord.push_back(j);
                coord.push_back(0);
                pegs.push_back(coord);
            }
        }
    }

    return pegs;
}
void GameBoard::euroConfig_Start()//Standard Configuration config
//36 1's , 1 0 and 12 (-1)'s
{
    for (int i = 0; i < (int) board.size(); ++i)
    {
        for (int j = 0; j < (int) board[i].size(); ++j)
        {
            if ((i == 0) && (j == 2))
            {
                board[i][j] = 0;
            } else if (board[i][j] == 0)
            {
                board[i][j] = 1;
            }
        }
    }
}

void GameBoard::euroConfig_Random()// creates a random state with random number of pegs
      .
{
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_int_distribution<int> distribution(1, 100);

    for (int i = 0; i < (int) board.size(); ++i)
    {
        for (int j = 0; j < (int) board[i].size(); ++j)
        {
            if (board[i][j] == 0)
```

```
            {
                int randNum = distribution(mt);
                board[i][j] = randNum % 2;
            }
        }
    }
}

bool GameBoard::makeMove(int id, int r, int c) //Will make the move. MUST RUN
    checkIfMoveValid OR SEG-FAULTS!!!!
{
    switch (id)
    {
    case 0:  {// up
        board[r][c] = 0;
        board[r - 1][c] = 0;
        board[r - 2][c] = 1;
        return true;
        break;
    }
    case 1:  {//right
        board[r][c] = 0;
        board[r][c + 1] = 0;
        board[r][c + 2] = 1;
        return true;
        break;
    }
    case 2:  {// down
        board[r][c] = 0;
        board[r + 1][c] = 0;
        board[r + 2][c] = 1;
        return true;
        break;
    }
    case 3:  {//left
        board[r][c] = 0;
        board[r][c - 1] = 0;
        board[r][c - 2] = 1;
        return true;
        break;
    }
    default: {
        std::cout << "Invalid ID \n";
        return false;
        break;
    }

    }

}
bool GameBoard::checkIfMoveValid(int id, int r, int c) //Given a direction and a pair of
    coodinates
// will check if that move would be valid
{
    switch (id)
    {
    case 0: {    //up
```

```
        if ((r > 1) && (board[r - 2][c] == 0) && (board[r - 1][c] == 1) && (board[r][c] == 1)
            ) // last check might be redundant but be safe
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    case 1: {//right
        if ((c < 5) && (board[r][c + 2] == 0) && (board[r][c + 1] == 1) && (board[r][c] == 1)
            )
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    case 2: {//down
        if ((r < 5) && (board[r + 2][c] == 0) && (board[r + 1][c] == 1) && (board[r][c] == 1)
            )
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    case 3: {//left
        if ((c > 1) && (board[r][c - 2] == 0) && (board[r][c - 1] == 1) && (board[r][c] == 1)
            )
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    default: {
        std::cout << "Invalid ID \n";
        return false;
        break;
    }
    }
}

bool GameBoard::checkIfMoveValidReverse(int id, int r, int c) //Given a direction and a pair
    of coodinates
// will check if that reverse move would be valid
```

```
{
    switch (id)
    {
    case 0: {    //up
        if ((r > 1) && (board[r − 2][c] == 1) && (board[r − 1][c] == 0) && (board[r][c] == 0)
            ) // last check might be redundant but be safe
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    case 1: {//right
        if ((c < 5) && (board[r][c + 2] == 1) && (board[r][c + 1] == 0) && (board[r][c] == 0)
            )
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    case 2: {//down
        if ((r < 5) && (board[r + 2][c] == 1) && (board[r + 1][c] == 0) && (board[r][c] == 0)
            )
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    case 3: {//left
        if ((c > 1) && (board[r][c − 2] == 1) && (board[r][c − 1] == 0) && (board[r][c] == 0)
            )
        {
            return true;
            break;
        } else
        {
            return false;
            break;
        }
    }
    default: {
        std::cout << "Invalid ID \n";
        return false;
        break;
    }
    }
}
```

```
void GameBoard::copy(GameBoard gb)
{
    this->board = gb.board;
}

bool GameBoard::equals(GameBoard gb)
{
    bool isEqual = true;
    if (this->board.size() != gb.board.size())
        return false;

    for (int i = 0; i < row; i++)
    {
        if (this->board[i].size() != gb.board[i].size())
            return false;

        isEqual = std::equal(this->board[i].begin(), this->board[i].end(), gb.board[i].begin());
        if (isEqual == false)
            return false;
    }

    return isEqual;
}

void GameBoard::getMoves(std::vector<Move> &path)//pushes all possible moves of valid pegs
    onto the path stack.
{
    for (int i = 0; i < row; i++)
    {
        for (int j = 0; j < col; j++)
        {
            if (board[i][j] == 1)
            {
                int id = 0;
                Move possible_move_up(id, i, j);
                path.push_back(possible_move_up);

                id = 2;
                Move possible_move_down(id, i, j);
                path.push_back(possible_move_down);

                id = 3;
                Move possible_move_left(id, i, j);
                path.push_back(possible_move_left);

                id = 1;
                Move possible_move_right(id, i, j);
                path.push_back(possible_move_right);
            }
        }
    }
}

bool GameBoard::checkGameEnd()//Checks the 4 positions (above, below, to the left, to the
    right) of a peg for another peg.
//If one exists then we havnt reached an end state yet as there is still a move possible.
{
```

```
    std::vector<std::vector<int>> pegs = this->getPegs();
    for (int i = 0; i < (int) pegs.size(); ++i)//for each peg
    {
        if (pegs[i][0] > 0) //can check up
            if (board[pegs[i][0] - 1][pegs[i][1]] == 1)
                return false;

        if (pegs[i][0] < 6) //can check down
            if (board[pegs[i][0] + 1][pegs[i][1]] == 1)
                return false;

        if (pegs[i][1] > 0) //can check left
            if (board[pegs[i][0]][pegs[i][1] - 1] == 1)
                return false;

        if (pegs[i][1] < 6) //can check right
            if (board[pegs[i][0]][pegs[i][1] + 1] == 1)
                return false;
    }
    return true;
}

bool GameBoard::checkGameWin()
{
    std::vector<std::vector<int>> pegs = this->getPegs();

    // end states for European
    // 3:   0,2
    //      1,3
    //      2,3
    bool state1 = pegs.size() == 1 && this->board[0][2] == 1;
    bool state2 = pegs.size() == 1 && this->board[1][3] == 1;
    bool state3 = pegs.size() == 1 && this->board[2][3] == 1;

    // cout<<"State 1: " <<state1<<"State 2: " <<state2<<"State 3: " <<state3<<endl;

    if (state1 || state2 || state3)
    {
        return true;
    }

    return false;
}

bool GameBoard::makeReverseMove(int id, int r, int c)
{
    switch (id)
    {
    case 0: {// up
        board[r][c] = 1;
        board[r - 1][c] = 1;
        board[r - 2][c] = 0;
        return true;
        break;
    }
    case 1: {//right
        board[r][c] = 1;
        board[r][c + 1] = 1;
```

```
        board[r][c + 2] = 0;
        return true;
        break;
    }
    case 2:  {// down
        board[r][c] = 1;
        board[r + 1][c] = 1;
        board[r + 2][c] = 0;
        return true;
        break;
    }
    case 3:  {//left
        board[r][c] = 1;
        board[r][c - 1] = 1;
        board[r][c - 2] = 0;
        return true;
        break;
    }
    default: {
        std::cout << "Invalid ID \n";
        return false;
        break;
    }

    }
}
```

8.3 Source: move.h

```
class Move
{
private:

public:
    Move(int id, int x, int y);

    int id;
    int r;
    int c;
};

Move::Move(int id, int x, int y)
{
    this->id = id;
    this->r = x;
    this->c = y;
}
```

8.4 Source: main.cpp

```
/*
AAA Assignment 2017

Evan Bancroft 879192
Jason Chalom 711985
```

```
*/

#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include <omp.h>
#include <time.h>
#include <cmath>
#include <chrono>
#include "omp.h"

#include "move.h"
#include "helpers.cpp"
#include "solitaire_board.h"
#include "backtracking.cpp"


/* Global variables */
#define app_name "COMS3005 Assignment 2017"
#define results1_header "amount,path length,time,found"
#define results1_location "./results/results_exp1_stack.csv"
#define results2_header "amount,path length,time,found,end state"
#define results2_location "./results/results_exp2_recurse.csv"
#define results3_header "amount,time,end state"
#define results3_location "./results/results_exp3_recurse.csv"
GameBoard gb;

/* Headers */
int main(int argc, char *argv[]);
void test();
void printPath(vector<vector<int>> path);
void run_stack_backtracking();
void run_recursive_backtracking();
void process_args(int argc, char *argv[]);


using namespace std;
int main(int argc, char *argv[])
{
    print_cmd_heading(app_name);
    process_args(argc, argv);


    /*Board testing*/
    if (argc == 1)
    {
        print_usage(argv);
        halt_execution();
    }

    return EXIT_SUCCESS;
}

void test()
{
    int numValidMoves = 0;
    int totalNumPegs = 0;
    cout << "TESTS...." << endl;
```

```
    GameBoard gb_new;
    //gb_new.euroConfig_Start();
    gb_new.board[2][2] = 1;
    gb_new.board[3][2] = 1;
    gb_new.board[4][3] = 1;
    gb_new.board[5][4] = 1;
    gb_new.board[3][5] = 1;
    cout << "Result: " << endl;
    gb_new.printBoard();

    std::vector<std::vector<int>> path;
    gb_new = backtracking_stack(gb_new, path, totalNumPegs, numValidMoves);
    bool found = gb_new.checkGameWin();
    gb_new.printBoard();
    int numPegs = gb_new.numPegs();
    cout << "amount: " << numPegs << " path_length: " << path.size() << endl << "
        Found: " << found << endl;
    printPath(path);
}

void runBestCase(int num)
{
    cout << "Running experiment 3...\n\n";

    int numValidMoves = 0;
    int totalNumPegs = 0;
    write_results_to_file(results3_location, results3_header, "");

    double total_start = omp_get_wtime();

    for (int i = 1; i <= 36; i = i + 1)
    {
        totalNumPegs = 0;
        int amount = i;
        GameBoard bc = bestCase(amount);;
        std::vector<std::vector<int>> path;
        double start = omp_get_wtime();

        // Add what ever being timed here
        bc = backtracking_stack(bc, path, totalNumPegs, numValidMoves);
        double time = omp_get_wtime() - start;

        bool found = bc.checkGameWin();
        // Output results
        cout << "amount: " << amount << " path_length: " << path.size() << " time: " <<
            time << " Found: " << found << endl;

        double avgNumPegs = 0.0;
        if (numValidMoves != 0)
        {
            avgNumPegs = (double)totalNumPegs / numValidMoves;
        }
        cout << "AverageNumPegs: " << avgNumPegs << endl;
        cout << "totalNumPegs: " << totalNumPegs << "\t numValidMoves: " <<
            numValidMoves << endl << endl;

        // print file line
        ostringstream out;
```

```
        out << amount << "," << path.size() << "," << time << "," << found << endl;
        write_results_to_file(results3_location, out.str());
    }

}

void printPath(vector<vector<int>> path)
{
    for (int i = 0; i < (int)path.size(); ++i)
    {
        cout << "I: " << i << '\t' << "(" << path[i][2] << "," << path[i][0] << "," <<
            path[i][1] << ")" << std::endl;
    }
}

void run_stack_backtracking()
{
    cout << "Running experiment 1...\n\n";
    int numValidMoves = 0;
    int totalNumPegs = 0;
    write_results_to_file(results1_location, results1_header, "");

    double total_start = omp_get_wtime();

    for (int i = 1; i <= 36; i = i + 1)
    {
        totalNumPegs = 0;
        int amount = i;
        GameBoard gb_new(i);
        std::vector<std::vector<int>> path;
        double start = omp_get_wtime();

        // Add what ever being timed here
        gb_new = backtracking_stack(gb_new, path, totalNumPegs, numValidMoves);
        double time = omp_get_wtime() - start;

        bool found = gb_new.checkGameWin();

        // Output results
        cout << "amount: " << amount << " path_length: " << path.size() << " time: " <<
             time << " Found: " << found << endl;

        double avgNumPegs = 0.0;
        if (numValidMoves != 0)
        {
            avgNumPegs = (double)totalNumPegs / numValidMoves;
        }
        cout << "AverageNumPegs: " << avgNumPegs << endl;
        cout << "totalNumPegs: " << totalNumPegs << "\t numValidMoves: " <<
            numValidMoves << endl << endl;

        // print file line
        ostringstream out;
        out << amount << "," << path.size() << "," << time << "," << avgNumPegs <<
            endl;
        write_results_to_file(results1_location, out.str());
    }
```

```
    double total_time = omp_get_wtime() - total_start;
    cout << "\n\ntotal time: " << total_time << " seconds." << endl;
}

void run_recursive_backtracking()
{
    cout << "Running experiment 2 (recursive)...\n\n";
    write_results_to_file(results2_location, results2_header, "");

    double total_start = omp_get_wtime();
    GameBoard final_1;
    final_1.board[0][2] = 1;

    GameBoard final_2;
    final_2.board[1][3] = 1;

    GameBoard final_3;
    final_3.board[2][3] = 1;

    for (int i = 1; i <= 17; i = i + 1)
    {
        int amount = i;
        std::vector<Move> path;
        bool found = false;
        int end_state = 1;
        double start = 0.0, time = 0.0;

        GameBoard gb_new(amount);
        start = omp_get_wtime();
        // Add what ever being timed here
        found = backtracking_recursive(gb_new, final_1, path);
        time = omp_get_wtime() - start;

        if (found == false)
        {
            gb_new = GameBoard(amount);
            path = std::vector<Move>();
            start = omp_get_wtime();
            // Add what ever being timed here
            found = backtracking_recursive(gb_new, final_2, path);
            time = omp_get_wtime() - start;
            end_state = 2;
        }

        if (found == false)
        {
            gb_new = GameBoard(amount);
            path = std::vector<Move>();
            start = omp_get_wtime();
            // Add what ever being timed here
            found = backtracking_recursive(gb_new, final_3, path);
            time = omp_get_wtime() - start;
            end_state = 3;
        }

        // Output results
        // "amount,number_denominations,time,found,end state"
```

```
        cout << "amount: " << amount << " path_length: " << path.size() << " time: " <<
            time << " Found: " << found << " End State: " << end_state << endl <<
            endl;

        // print file line
        ostringstream out;
        out << amount << "," << path.size() << "," << time << "," << found << "," <<
            end_state << endl;
        write_results_to_file(results2_location, out.str());
    }

    double total_time = omp_get_wtime() - total_start;
    cout << "\n\ntotal time: " << total_time << " seconds." << endl;
}

void process_args(int argc, char *argv[])
{
    for (int i = 1; i < argc; i++)
    {
        string str = string(argv[i]);
        if (contains_string(str, "h") || contains_string(str, "help"))
        {
            print_usage(argv);
            halt_execution();
        }

        if (contains_string(str, "-t") || contains_string(str, "tests"))
        {
            test();
        }

        if (contains_string(str, "-rf") || contains_string(str, "run_full"))
        {
            gb.euroConfig_Start();
            gb.printBoard();
        }

        if (contains_string(str, "-rr") || contains_string(str, "run_rand") || contains_string
            (str, "runr"))
        {
            gb.euroConfig_Random();
            gb.printBoard();
        }

        if (contains_string(str, "-rb") || contains_string(str, "run_back") || contains_string
            (str, "runb"))
        {
            run_stack_backtracking();
        }
        if (contains_string(str, "-bc"))
        {
            runBestCase(24);
        }

        if (contains_string(str, "-recurse"))
        {
            run_recursive_backtracking();
        }
```

```
        if (contains_string(str, "-m") || contains_string(str, "manual"))
        {

            int id = 0;
            int x = 0;
            int y = 0;
            cin >> id;
            while (id != -1)
            {
                cin >> x;
                cin >> y;
                if (gb.checkIfMoveValid(id, x, y))
                {
                    cout << gb.makeMove(id, x, y) << '\n';
                }
                gb.printBoard();
                cout << "Num Pegs:" << gb.numPegs() << '\n';
                cin >> id;
            }

        }

    }
}
```

8.5 Source: helpers.cpp

```
/*Jason Chalom 711985
   Helper Functions 2017
*/

/*Random generator for c++11*/
/* std::mt19937 rng;
    rng.seed(std::random_device()());
    std::uniform_int_distribution<std::mt19937::result_type> dist6(0,n); // distribution in
         range [1, 6]*/

#include <stdio.h>
#include <stdlib.h>
#include <cmath>
#include <time.h>
#include <string>
#include <fstream>
#include <sstream>
#include <random>
#include <algorithm>
#include "omp.h"

using namespace std;

/*MISC*/
int random_index(int a, int b);
void halt_execution(string message);
void write_results_to_file (std::string filename, std::string results);
void write_results_to_file (std::string filename, std::string header, std::string results);
void print_cmd_heading(string app_name);
```

```
void print_usage(char *argv[]);

/*MISC*/
int random_index(int a, int b)
{
    std::random_device rd;
    std::mt19937 rng(rd());
    std::uniform_int_distribution<int> uni(a, b);

    return uni(rng);
}

void halt_execution(string message = "")
{
    cout << message << endl;
    exit(EXIT_FAILURE);
}

void write_results_to_file(std::string filename, std::string results)
{
    ofstream file;
    file.open(filename.c_str(), ios::app);
    file << results;
    file.close();
}

void write_results_to_file(std::string filename, std::string header, std::string results)
{
    ofstream file;
    file.open(filename.c_str(), ios::app);
    file << header << results << endl;
    file.close();
}

void print_cmd_heading(string app_name)
{
    printf("%s\nJason Chalom 711985\nEvan Bancroft 879192\n2017\n\n", app_name.c_str());
}

void print_usage(char *argv[])
{
    printf("At least two parameters must be selected.\n\n");
    printf("usage: %s -rr -m\n", argv[0]);
    printf("Random state -rr\n");
    printf("Full state -rf\n");
    printf("Run Stacked Based Backtracking -rb\n");
    printf("Run Recursive Backtracking -recurse\n");
    printf("Manual -r\n");
    printf("Help -h\n");
}

bool contains_string(string input, string str)
{
    if (input.find(str) != string::npos) {
        return true;
    }
    return false;
```

```
}
```

Appendix B: Results

Amount	Time(Best Case states)	Time(Random states)	Found A Path?	End State
1	0,000012993	0,000007902	0	3
2	0,000013994	0,000007642	0	3
3	0,000014695	0,00000827	0	3
4	0,000045349	0,000008324	0	3
5	0,000017047	0,00006715	0	3
6	0,000017154	0,000045409	0	3
7	0,000035677	0,000019492	0	3
8	0,000136504	0,000223362	0	3
9	0,000953186	0,00031693	0	3
10	0,0377231	0,0249256	0	3
11	0,578075	0,000131526	0	3
12	0,165076	0,00613507	0	3
13	0,00160808	1,3081	0	3
14	2,05269	0,00337623	0	3
15	9,56415	83,1621	1	3
16	0,228359	18,762	0	3
17	314,24	2442,48	0	3

Table 2: Timed Results of Recursive Implementation

Amount	Path Size	Time (Random State)	Time(Best Case State)
1	0	0,000087546	0,000079554
2	1	0,000180635	0,000087919
3	2	0,000088643	0,000111731
4	3	0,000089779	0,000135486
5	4	0,000326819	0,000147682
6	6	0,000095684	0,000334714
7	8	0,000097001	0,000492098
8	9	0,00132629	0,000701262
9	10	0,00198162	0,00102037
10	11	0,00194504	0,00138001
11	12	0,00236243	0,00171943
12	14	0,00278941	0,0016969
13	15	0,00338771	0,00189975
14	16	0,00449822	0,00250172
15	18	0,00444476	0,00323355
16	19	0,00549958	0,00417548
17	20	0,00680899	0,00539972
18	20	0,00917923	0,00540915
19	20	0,00908114	0,00544882
20	20	0,010037	0,00536063
21	20	0,0111601	0,00533272
22	20	0,0128546	0,00537615
23	20	0,0170417	0,00538194
24	20	0,0175451	0,00548215
25	20	0,0212161	0,00533657
26	20	0,0195111	0,00539675
27	20	0,0194284	0,00534268
28	20	0,0238981	0,00536282
29	20	0,0263172	0,00542806
30	20	0,0252291	0,00536498
31	20	0,0283234	0,00533903
32	20	0,0319084	0,00535362
33	20	0,030172	0,00537846
34	20	0,0382437	0,0053947
35	20	0,0450741	0,0053488
36	20	0,0423465	0,00541166

Table 3: Timed Results of Stack Based Implementation

Amount	**Average Moves Per Peg (Random case)**	**Average Moves Per Peg (Best case State)**
1	0	0
2	0	4
3	0	2,66667
4	0	2
5	4	1,4
6	4	1,8125
7	3,71429	1,91667
8	3,52941	2,0303
9	3,75	2,27907
10	4	2,37037
11	3,57895	2,39394
12	3,5493	1,875
13	3,37349	1,72632
14	3,20833	1,86486
15	2,89655	1,95349
16	3,375	2,12838
17	3,22759	2,2619
18	3,16981	2,02128
19	2,85057	1,82692
20	3,38798	1,66667
21	3,86667	1,53226
22	3,74775	1,41791
23	3,8247	1,31944
24	3,45763	1,23377
25	3,6129	1,15854
26	3,64308	1,09195
27	3,98834	1,03261
28	3,89474	0,979381
29	3,71163	0,931373
30	3,46309	0,88785
31	3,79381	0,848214
32	3,69925	0,811966
33	3,55862	0,778689
34	3,49762	0,748031
35	3,47977	0,719697
36	3,28937	0,693431

Table 4: Average Available Moves Per Peg for the Stack Implementation