Advanced Analysis of Algorithms Assignment

Tau Merand 908096 Vincent Varkevisser 705668

October 31, 2017

Abstract

The game of peg solitaire is a NP complete game. In this project we set out to implement a fast, efficient method to find optimum solutions to the English version of this game. Further analysis as well as experimentation was used to test the validity of the solution that was created and to explore the computational complexity of the game. The code written for this project is available at https://github.com/vapour101/ParallelProject

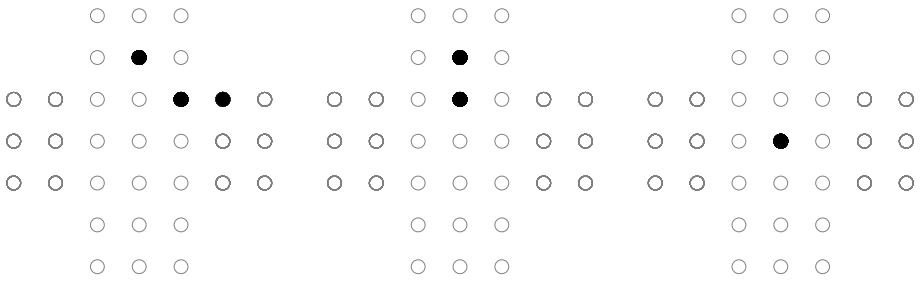
UNIVERSITY OF THE
WITWATERSRAND,
JOHANNESBURG

Contents

1 Introduction

The game of peg solitaire is a one player game played on a 33 holed cross shaped board that involves jumping pegs over other pegs, in a manner similar to checkers. The rules are as follows:

1. A move consists of jumping a peg over an orthogonal neighbor into an empty space. The peg that was jumped over is then removed from the board.
2. Pegs can only jump onto an empty space.
3. The game is won if the final peg is in the centre space.
4. If no pegs can legally move or the final peg is not in the centre the game is lost.



(a) A start state

(b) The right most peg jumps over its left neighbor

(c) The winning state

Figure 1: A winning set of valid moves

2 Solution Technique

Recursive backtracking using depth first search was chosen as the method for state space exploration. Unlike the brute force approach to finding a solution to a game, the game tree is not fully precomputed and then explored. Instead in Algorithm 1 the tree is generated by choosing a random legal move performing it and then evaluating the resultant state. If the resulting state is not the winning state a random legal move is chosen and performed on that state. This is repeated until a state with no legal moves is reached. If that terminal state is the winning state then the sequence of moves taken to reach that state is returned. If it is not the winning state then the algorithm backtracks to the previous state and choses a different move. Thus the game tree is generated in a depth first manner as it is explored. Due to the nature of peg solitaire it is sufficient to return the first solution found as the optimal solution (see Algorithmic Analysis)

```
1  Function backtrack
       input  : An initial, possibly un-winnable, board state.
       output: A sequence of moves to get from the initial state to the
                winning state, if a winning state cannot be reached the
                sequence should be empty.
2      initialState ← The initial state
3      legalMoves ← A list of all the legal moves for initialState
4      result ← Empty list to store moves from the initial state to the
        winning state
5      foreach move in legalMoves do
6          state ← The state after playing move on initialState
7          if state is a winning state then
8              result ← move
9              break
10         end
11         childResult ← backtrack(state)
12         if childResult is not empty then
13             Prepend move to childResult
14             result ← childResult
15             break
16         end
17     end
18     return result
```

Algorithm 1: A standard recursive backtracking using DFS

But because pegs are indistinguishable, game states where pegs are in the same position are identical, regardless of the moves taken to arrive at that state. Thus exploring the state space looking for a sequence of states leading to the winning state will probably involve evaluating the same states many times. Thus a significant speed up can be achieved by saving states that are known to not lead to the winning state as shown in Algorithm 2. A hash set of infeasible states is kept as the game tree is explored with states inserted into this set if it is not a winning state and there are no legal moves from that set or if all of its children are already in the set.

```
1  Function dynamicBacktrack
       input  : An initial, possibly un-winnable, board state.
       input  : A set of all states that have been searched and are known to
                be un-winnable.
       output: A sequence of moves to get from the initial state to the
                winning state, if a winning state cannot be reached the
                sequence should be empty.
2      initialState ← The initial state
3      infeasibleSet ← The set of un-winnable states
4      legalMoves ← A list of all the legal moves for initialState
5      result ← Empty list to store moves from the initial state to the
         winning state
6      foreach move in legalMoves do
7          state ← The state after playing move on initialState
8          if state is in infeasibleSet then
9              continue
10         end
11         if state is a winning state then
12             result ← move
13             break
14         end
15         childResult ← dynamicBacktrack(state, infeasibleSet)
16         if childResult is not empty then
17             Prepend move to childResult
18             result ← childResult
19             break
20         else
21             Add state to infeasibleSet
22         end
23     end
24     return result
```

Algorithm 2: Recursive backtracking using DFS and dynamic programming methods

3 Analysis

3.1 Algorithmic Analysis

For any given state S let P be the number of pegs and E be the number of empty spaces. There are 33 spaces on a board so $E = 33 - P$
For any legal move a peg must jump over an orthogonal neighbour and that neighbour is then removed. Therefore for any initial state a solution must consist of exactly $P - 1$ moves. If more moves were performed then there would be no pegs left and if fewer were performed the resultant board wouldn't be the final state. The best case for a depth first algorithm is going to be if the correct move was selected and performed at each level of the tree, thus the best case for our backtracking DFS, with or without the dynamic programming methods, would be $O(P - 1) = O(P)$
On the other hand the worst case would be when the entire game tree has to be explored to find a solution. One can estimate an upper bound on the number of legal moves that can be made in any given game state in two ways. Firstly pegs have to jump orthogonally so each peg could jump, at most, in 4 directions. Alternatively since pegs have to jump into empty spaces, each empty space could be jumped into by at most 4 pegs. For a state that has 32 pegs and 1 empty space, e.g. the standard start state, there only 4 legal moves. Whereas in a state with 2 neighbouring pegs and 31 empty spaces there are only 2 possible moves. Therefore an upper bound on the number of moves could be the minimum$\{4P, 4E\}$ but $4P < 4(33 - P)$ $\forall P \leq 16$ giving an upper bound of $4 \times 16 = 64 = 2^6$ on the number of moves that can be made in any state.
As discussed above the number of moves to find a solution is $P - 1$ but for some invalid paths the moves may lead to an invalid state that while $P > 1$, no legal moves can be performed due to the relative positions of the remaining pegs. Thus $P - 1$ is an upper bound on the depth with which any branch must explored, i.e. the maximum height of the game tree is $P - 1$.
If we consider the root node to be at depth 0 of the tree, at depth 1 there are at most 2^6 states, at depth 2 there are $(2^6)^2$ states. Following this pattern, at the maximum depth of $P - 1$ there are $(2^6)^{P-1}$ terminal states. Therefore the total number of states in the game tree is $1 + (2^6) + (2^6)^2 + (2^6)^3 + ... + (2^6)^{P-1} = \sum_{i=0}^{i=P-1} (2^6)^i = \frac{(2^6)^P - 1}{(2^6) - 1}$ Therefore the worst case would be $O(2^P)$ which is equivalent to $O(2^N)$ where N is the number of moves in the solution.

3.2 Empirical Analysis

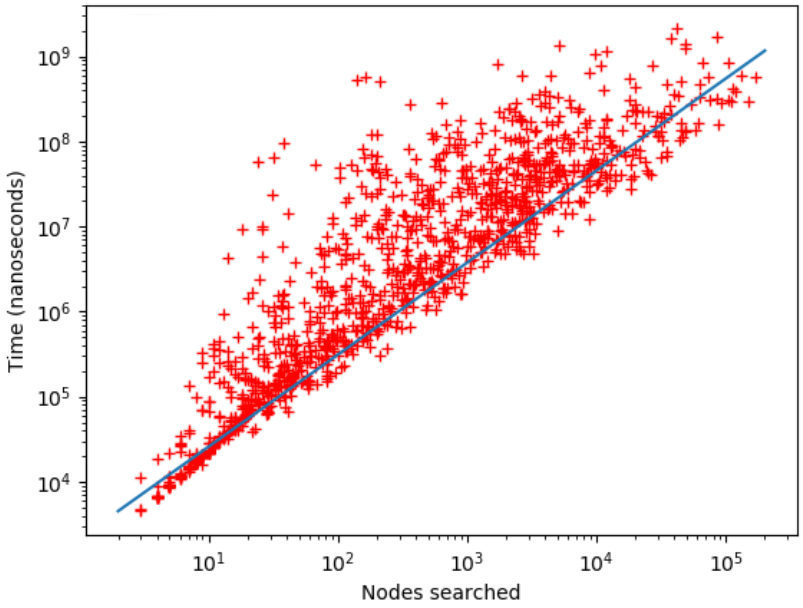


Figure 2: Graph comparing running time with the number of elements searched

Figure 2 shows that the running time can be fit to the line $2133.73x^{1.083}$ with an R^2 value of 0.81 meaning the algorithm is of $O(n^{0.81}) \approx O(n)$ complexity in the number of elements of the tree searched.

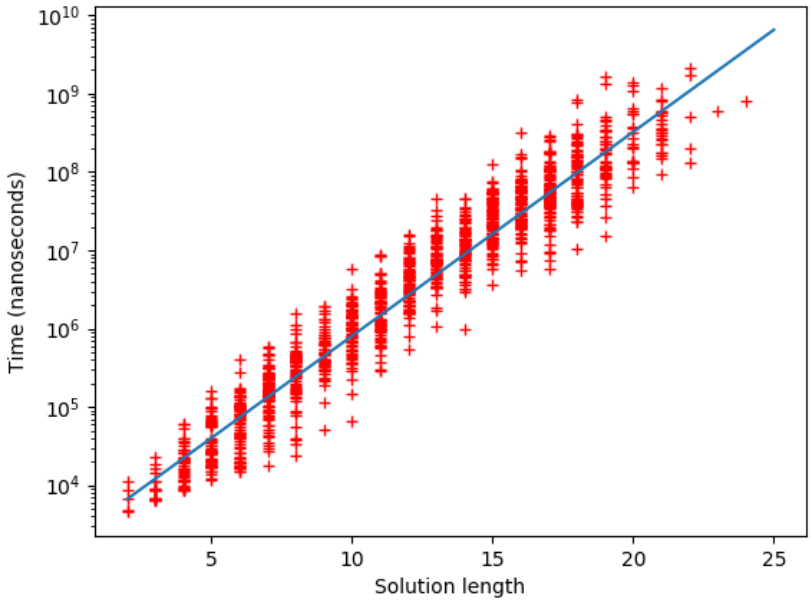


Figure 3: Graph comparing running time with the length of the solution found

Figure 3 shows that the running time can be fit to the line $2000e^{0.6x}$ meaning the algorithm is of $O(e^n)$ complexity in the length of the solution.
Each red cross in the above graphs represent a different start state that the solution technique was run on. Each start state was generated using valid moves in reverse from the winning state. As can be seen from Figure 2 touching each game state in the tree takes a constant time, hence the linear nature of that graph whereas Figure 3 shows that the total running time of the solution technique scales exponentially with the number of moves needed to reach the final state.

A Work Split

Vincent Varkevisser (705668) and Tau Merand (908096) pair programmed the solution technique implementation listed in Appendix B. Vincent then optimised this code as well as performed the timing and graphing of the experimennts and wrote the empirical analysis and pseudocode sections of this document. Tau Merand wrote the experimental generator as well as the introduction and algorithmic analysis of this document. It is our belief that we split the work quite fairly and equally.

B Code

Listing 1: backtrack.cpp

```
#include "backtrack.h"
#include <iostream>
#include <algorithm>

using namespace std;

BackTrack::BoardSequence BackTrack::BoardSequence::NullSequence;

BackTrack::MoveSequence BackTrack::recurse(const Board& state)
{
	nodes++;

	vector<Move> legal = state.getLegalMoves();

	if (legal.empty())
		failed++;

	if (legal.size() > 5)
		random_shuffle(legal.begin(), legal.end());

	for (Move move : legal)
	{
		Board privateState = state;
		privateState.executeMove(move);

		if (isInfeasible(privateState))
			continue;

		if ( privateState.isFinal() )
		{
			MoveSequence result;
```

```
                        result.push_front(move);
                        return result;
                }

                MoveSequence childResult = recurse(privateState);

                if ( !childResult.empty())
                {
                        childResult.push_front(move);
                        return childResult;
                }

                addInfeasible(privateState);
        }

        return MoveSequence();
}

ostream& operator<<(ostream& out, const BackTrack& solver)
{
        solver.printToStream(out);
        return out;
}

BackTrack::BackTrack(const Board& start) : initialBoard{start}
{
        checked = false;
        solvable = false;
        failed = 0;
        nodes = 0;
}

void BackTrack::start()
{
        chrono::high_resolution_clock::time_point start, stop;

        start = chrono::high_resolution_clock::now();
        solution = recurse(initialBoard);
        stop = chrono::high_resolution_clock::now();

        duration = stop - start;
        checked = true;
        solvable = !solution.empty();
}

BackTrack::BoardSequence BackTrack::getBoardSequence() const
```

```
{
	if ( !hasSolution() )
		return BoardSequence::NullSequence;

	return BoardSequence(*this);
}

BackTrack::MoveSequence BackTrack::getMoveSequence() const
{
	return solution;
}

BackTrack::SolutionTime BackTrack::getDuration() const
{
	if ( !checked )
		return SolutionTime::max();

	return duration;
}

int BackTrack::getInfeasibleCount() const
{
	return infeasibleBoards.size();
}

int BackTrack::getFailed() const
{
	return failed;
}

int BackTrack::getNodes() const
{
	return nodes;
}

bool BackTrack::hasSolution() const
{
	return solvable;
}

void BackTrack::printToStream(ostream& out) const
{
	out << initialBoard << endl;

	if ( !checked )
	{
```

```
                out << "Solution not started." << endl;
                return;
        }

        if (!hasSolution())
        {
                out << "No solution found." << endl;
                return;
        }

        out << "Solution found in " << getDuration() << endl;
        out << "Checked dead states: " << getFailed() << endl;
        out << "Infeasible states found: " << getInfeasibleCount()
            ↪ << endl;
        out << "Length: " << getMoveSequence().size() << " moves."
            ↪ << endl;
        out << getMoveSequence() << endl;
}

void BackTrack::printSequence(ostream& out) const
{
        if (!hasSolution())
                return;

        out << getBoardSequence() << endl;
}

void BackTrack::printTime(ostream& out) const
{
        out << "Time:\t\t";
        out << getDuration().count() / 1000000 << "ms" << endl;
}

void BackTrack::clear()
{
        checked = false;
        solvable = false;
        nodes = 0;
        solution.clear();
        infeasibleBoards.clear();
}

bool BackTrack::isInfeasible(const Board& check)
{
        string boardString = check.toString();
        return infeasibleBoards.count(boardString);
```

```
}

void BackTrack::addInfeasible(const Board& board)
{
	infeasibleBoards.insert(board.toString());
}


ostream& operator<<(ostream& out, const BackTrack::SolutionTime&
    ↪ duration)
{
	out << duration.count() / 1000000 << "ms";
	return out;
}

ostream& operator<<(ostream& out, const BackTrack::BoardSequence&
    ↪  solution)
{
	if (solution.isNullSequence)
		return out;

	Board board = solution.initialBoard;
	out << board << endl;

	for (Move move : solution.sequence)
	{
		board.executeMove(move);
		out << board << endl;
	}

	return out;
}

BackTrack::BoardSequence::BoardSequence(const BackTrack& solver)
    ↪ : isNullSequence{false}, initialBoard{solver.initialBoard
    ↪ }, sequence{solver.solution}
{
}

BackTrack::BoardSequence::BoardSequence() : isNullSequence{true}
{
}

ostream& operator<<(ostream& out, const BackTrack::MoveSequence&
    ↪ sequence)
{
```

```
        out << "Move␣Sequence:" << endl;
        for (Move move : sequence)
        {
                out << "\t";
                printMove(out, move);
        }

        return out;
}
```

Listing 2: backtrack.h

```
#ifndef BACKTRACK_H
#define BACKTRACK_H

#include <list>
#include <unordered_set>
#include <ostream>
#include <chrono>

#include "board.h"

class BackTrack
{
public:
	class BoardSequence; //forward declaration
	typedef std::chrono::nanoseconds SolutionTime;
	typedef std::list<Move> MoveSequence;

	BackTrack(const Board& start);

	void start();
	void clear();

	bool hasSolution() const;

	BoardSequence getBoardSequence() const;
	MoveSequence getMoveSequence() const;
	SolutionTime getDuration() const;
	int getInfeasibleCount() const;
	int getFailed() const;
	int getNodes() const;


	void printSequence(std::ostream& out) const;
	void printTime(std::ostream& out) const;

private:
	void printToStream(std::ostream& out) const;
	bool isInfeasible(const Board& check);
	void addInfeasible(const Board& board);
	MoveSequence recurse(const Board& state);

	const Board initialBoard;
	int failed;
	int nodes;
```

```
        bool checked;
        bool solvable;
        MoveSequence solution;

        std::unordered_set<std::string> infeasibleBoards;
        SolutionTime duration;

        friend std::ostream& operator<<(std::ostream& out, const
            ↪ BackTrack& solver);

public:
        class BoardSequence {
        public:
                BoardSequence(const BackTrack& solver);

                const bool isNullSequence;

                static BoardSequence NullSequence;

                friend std::ostream& operator<<(std::ostream& out,
                    ↪ const BoardSequence& sequence);

        private:
                BoardSequence();
                const Board initialBoard;
                const std::list<Move> sequence;
        };
};

std::ostream& operator<<(std::ostream& out, const BackTrack::
    ↪ SolutionTime& duration);
std::ostream& operator<<(std::ostream& out, const BackTrack::
    ↪ MoveSequence& sequence);


#endif // BACKTRACK_H
```

Listing 3: board.cpp

```
#include "board.h"
#include "cmath"
#include <vector>
#include <algorithm>

using namespace std;

ostream& operator<<(ostream& out, const Board& board)
{
	out << board.toHumanString();
	return out;
}

Board::Board()
{
	for (int i = 0; i < 7; i++)
		for (int j = 0; j < 7; j++)
		{
			if (i == 3 && j == 3)
				continue;

			Coord coord = make_pair(i, j);
			if (withinBounds(coord))
				pegs.insert(coord);
		}
}

Board::Board(const Board& other)
{
	for (Coord peg : other.pegs) {
		pegs.insert(peg);
	}
}

Board::Board(const string& input)
{
	char temp;
	istringstream inStream{input};

	for (int j = 6; j >= 0; j--)
	{
		for (int i = 0; i < 7; i++)
		{
			Coord pos = make_pair(i, j);
```

```
                        inStream >> temp;

                        if (withinBounds(pos))
                        {
                                if (temp == 'o')
                                        pegs.insert(pos);
                                else if (temp != '+')
                                        throw "Parse error";
                        }
                        else {
                                if (temp != '.')
                                        throw "Parse error";
                        }
                }
        }
}

bool Board::isFinal() const
{
        return pegs.size() == 1 && pegs.count(make_pair(3,3));
}

bool Board::isLegal(Move move) const
{
        if (!isOccupied(move.first))
                return false;

        Coord pos = getPosition(move);

        if (!withinBounds(pos))
                return false;

        if (isOccupied(pos))
                return false;

        if (!captures(move))
                return false;

        return true;
}

void Board::executeMove(Move move)
{
        if (!isLegal(move))
                return;
```

```
        pegs.erase(pegs.find(move.first));
        pegs.erase(pegs.find(getJumped(move)));

        pegs.insert(getPosition(move));
}

string Board::toString() const
{
        string out = "";

        for (int j = 6; j >= 0; j--)
        {
                for (int i = 0; i < 7; i++)
                {
                        Coord coord = make_pair(i, j);
                        if (withinBounds(coord))
                        {
                                if (isOccupied(coord))
                                        out += "o";
                                else
                                        out += "+";
                        }
                        else
                                out += ".";
                }
        }

        return out;
}

vector<string> Board::toImageStrings() const
{
        vector<string> out = {"", "", "", "", "", "", "", ""};

        for (int j = 6; j >= 0; j--)
        {
                for (int i = 0; i < 7; i++)
                {
                        Coord coord = make_pair(i, j);

                        for (int k = 0; k < 8; k++)
                        {
                                Coord transCoord = transformCoord(
                                    ↪ coord, k);

                                if (withinBounds(transCoord))
```

```
                        {
                                if (isOccupied(transCoord))
                                        out[k] += "o";
                                else
                                        out[k] += "+";
                        }
                        else
                                out[k] += ".";
                }
        }
    }

    return out;
}

bool Board::isInvalid() const
{
    for (const auto& coord: pegs){
        for (int i=0; i<=3; i++){
            Move move=std::make_pair(coord, (Direction)
                ↪ i);
            if (isLegal(move)){
                return false;
            }
        }
    }
    return true;
}

std::vector<Move> Board::getLegalMoves() const
{
    std::vector<Move> res{};

    for (Coord peg : pegs) {
        for (int i = 0; i < 4; ++i)
        {
            Move move=std::make_pair(peg, (Direction) i)
                ↪ ;

            if (isLegal(move)){
                res.push_back(move);
            }
        }
    }

    return res;
```

```
}

std::vector<size_t> Board::getImageHashes() const
{
	vector<size_t> out;

	for (string board : toImageStrings())
		out.push_back(hash<string>()(board));

	return out;
}

Coord Board::getPosition(Move move) const
{
	Coord coords = move.first;

	switch(move.second) {
	case NORTH:
		coords.second += 2;
		break;
	case SOUTH:
		coords.second -= 2;
		break;
	case EAST:
		coords.first += 2;
		break;
	case WEST:
		coords.first -= 2;
		break;
	}

	return coords;
}

bool Board::withinBounds(Coord coords) const
{
	int x = coords.first;
	int y = coords.second;

	if (x > 3)
		x = 6 - x;

	if (y > 3)
		y = 6 - y;

	if ( x < 0 || y < 0 )
```

```
                return false;

        if ( x < 2 && y < 2 )
                return false;

        return true;
}

bool Board::isOccupied(Coord coords) const
{
        return pegs.count(coords);
}

bool Board::captures(Move move) const
{
        return isOccupied(getJumped(move));
}

Coord Board::getJumped(Move move) const
{
        Coord start = move.first;

        switch(move.second) {
        case NORTH:
                start.second += 1;
                break;
        case SOUTH:
                start.second -= 1;
                break;
        case EAST:
                start.first += 1;
                break;
        case WEST:
                start.first -= 1;
                break;
        }

        return start;
}

string Board::toHumanString() const
{
        string out = "";

        for (int j = 6; j >= 0; j--)
        {
```

```
                for (int i = 0; i < 7; i++)
                {
                        Coord coord = make_pair(i, j);
                        if (withinBounds(coord))
                        {
                                if (isOccupied(coord))
                                        out += "o";
                                else
                                        out += "+";
                        }
                        else
                                out += ".";
                }

                out += "\n";
        }

        return out;
}


void printMove(ostream& out, Move move)
{
        printCoords(out, move.first);
        out << "␣-␣";
        printDirection(out, move.second);
        out << endl;
}

void printDirection(ostream& out, Direction direction)
{
        switch (direction) {
        case NORTH:
                out << "NORTH";
                break;
        case SOUTH:
                out << "SOUTH";
                break;
        case WEST:
                out << "WEST";
                break;
        case EAST:
                out << "EAST";
                break;
        }
}
```

```
void printCoords(ostream& out, Coord coord)
{
	out << "(" << coord.first << ", " << coord.second << ")";
}

Coord transformCoord(const Coord& in, int angle)
{
	angle %= 8;

	int x = in.first;
	int y = in.second;

	if ( angle & 1 ) //flip
		x = 6 - x;

	if ( angle & 2 ) //rotate 90*
	{
		int temp = x;
		x = 6 - y;
		y = temp;
	}

	if ( angle & 4 ) //rotate 180*
	{
		x = 6 - x;
		y = 6 - y;
	}

	return make_pair(x, y);


}
```

Listing 4: board.h

```
#ifndef BOARD_H
#define BOARD_H

#include <unordered_set>
#include <utility>
#include <boost/functional/hash.hpp>
#include <string>
#include <list>
#include <ostream>

struct PairHash {
public:
  template <typename T, typename U>
  std::size_t operator()(const std::pair<T, U> &x) const
  {
        std::size_t seed = 0;
        boost::hash_combine(seed, std::hash<T>()(x.first));
        boost::hash_combine(seed, std::hash<T>()(x.second));
        return seed;
  }
};

enum Direction {
    NORTH = 0, SOUTH = 1, EAST = 2, WEST = 3
};

typedef std::pair<int, int> Coord;
typedef std::pair<Coord, Direction> Move;

Coord transformCoord(const Coord& in, int angle);

class Board
{
public:
       Board();
       Board(const Board& other);
       Board(const std::string& input);

       bool isFinal() const;
       bool isLegal(Move) const;
       void executeMove(Move);
       std::string toString() const;
       std::vector<std::string> toImageStrings() const;
       bool isInvalid() const;
```

```
        std::vector<Move> getLegalMoves() const;
        std::vector<size_t> getImageHashes() const;

private:
        std::unordered_set<Coord, PairHash> pegs;
        Coord getPosition(Move) const;
        bool withinBounds(Coord) const;
        bool isOccupied(Coord) const;
        bool captures(Move) const;
        Coord getJumped(Move) const;
        std::string toHumanString() const;

        friend struct BoardHash;
        friend std::ostream& operator<<(std::ostream& out, const
            ↪ Board& board);
};


struct BoardHash {
public:
        std::size_t operator()(const Board& board) const
        {
                return std::hash<std::string>()(board.toString());
        }
};

void printDirection(std::ostream& out, Direction direction);
void printCoords(std::ostream& out, Coord coord);
void printMove(std::ostream& out, Move move);

#endif // BOARD_H
```