Heuristics in MCTS-based Computer Go

Can heuristics improve the performance of MCTS-based computer go?

FABIAN BERGMARK & JOHAN STENBERG

Stockholm 2014

Bachelor’s thesis DD143X
School of Computer Science and Communication
Kungliga Tekniska Högskolan
Mentor: Michael Minock
Authors contact: fbermark@kth.se, jostenbe@kth.se
Abstract

The subject of computer Go is an active field under AI and has achieved much attention in research. The current state of the art computer Go implementations use a game tree search approach rather than advanced heuristics. This thesis aims to bridge these two approaches and combine Monte Carlo Tree Search with heuristics to deduce if any general results can be found. The results of the thesis indicate that the performance of a combined MCTS-heuristic approach correlates strongly with performance of the heuristic. Furthermore, MCTS can be used with any heuristic to improve its performance.
## Contents

1 Introduction .................................................. 3

2 Background .................................................... 3
   2.1 Go ....................................................... 4
   2.2 Computer Go ........................................... 4
   2.3 Tree Search and Game Trees ............................... 4
   2.4 Monte Carlo ............................................. 5
   2.5 Monte Carlo Tree Search ................................. 5

3 Methods ....................................................... 6
   3.1 Framework Technical Details .............................. 6
   3.2 Endgame .................................................. 6
   3.3 Heuristic Implementations ................................. 7
   3.4 MCTS Implementation Details ............................. 7
   3.5 Evaluation ............................................... 8

4 Results ....................................................... 9
   4.1 Time Dependency Graph .................................. 9
   4.2 Comparison Result Tables ................................. 9
   4.3 Correlation Graph ....................................... 10

5 Conclusion ................................................... 11

6 Discussion .................................................... 11
   6.1 Comparison Limitations ................................. 11
   6.2 Performance against humans .............................. 11
   6.3 Implementation Weaknesses ............................... 12
   6.4 Choosing of Heuristics ................................. 12

7 References ................................................... 12

8 Appendix A .................................................... 14
1 Introduction

Go is a board game originating in China and first invented over 2,500 years ago\(^1\). Computer Go is a field under computer AI trying to implement a computer program that plays Go. To date, there is still considered to be a large gap between the best human players and the best AI playing the traditional 19x19 professional board\(^2\).

The subject of computer Go has received much attention in late years. During the 1990s research focused on implementing human expert knowledge and decision making combined with local search. This method have several drawbacks and today much research is focused on Monte Carlo algorithms\(^3\). These algorithms use randomness to prune the search space, only evaluating a limited set of plays. They are, in a sense, in the other end of the spectra using large search trees and randomness to find optimal moves\(^4\).

There exist much research about both methods, and a relevant question is thus if they can be combined in a constructive way.

Go differs from many other board game in the sense that it’s search space is seldom reduced when a player places another stone on the board\(^1\). Smart heuristics such as modeling an expert player or advanced local search simply requires, to be effective enough, a too large domain space. Instead, researchers are using Monte Carlo Tree Search\(^4\) (MCTS). MCTS is described more exhaustive in the background section but can be shortly summarized as follows; MCTS focuses on evaluating the most promising moves, basing the expansion of the game tree on random sampling of the current search space\(^5\). Since MCTS can be regarded as the opposite of using intelligent heuristics, our thesis researches, analyses and concludes if the two different approaches can be merged into a more successful one. Researchers have already visited this topic, and popular Computer Go implementations such as GNUGo also uses, in addition to MCTS, some hand-tuned heuristics\(^6\).

This thesis aims to bridge these two approaches and combine Monte Carlo Tree Search with heuristics to see if any general results can be found.

Many advocates for the Monte Carlo Tree Search points out that one of the algorithm’s strongest traits are that it is game-independent\(^5\). This thesis hopes to further enhance the MCTS for a domain specific usage, i.e. the field of computer Go. If the results are promising they could be correlated to other fields of usage for the MCTS.

2 Background

These sections shortly describe the key components of our background research.
2.1 Go

Go is played by two players which each places white and black stones on a grid\(^1\). Official games are played on a 19x19 board, but smaller dimensions are often used by beginners. Played stones can’t be moved but are captured if they are surrounded and are then removed from the board. The game is ended when neither player wants to make another move. The game of Go has multiple ways of deciding which player won a game, and this thesis uses stone scoring. In stone scoring, the player which the most stones and the end of the game is declared the winner.

One aspect of Go is the concept of Ko-rules. Ko rules dictates that a play is illegal if it would have the effect, after all steps of the play have been completed, of creating a position that has occurred previously in the game\(^1\). The consequence of this rule is that a player cannot recreate the board position from the player’s previous move.

2.2 Computer Go

Computer Go has been a research subject since the 1960s\(^3\). Early research focused on translating human expert knowledge into rules, thus implementing a tactical player. This approach suffers from the additive nature of Go resulting in a large search space. Algorithms were able to perform well in one, or several local areas but were unable to do board-spanning strategies\(^7\).

Recent research focuses on search trees, using statistical approximations and randomness to reduce the search space. Implementations using these methods have had some success against human players even on 19x19 boards\(^7\).

2.3 Tree Search and Game Trees

Tree search is a broad term when used within the field of computer science. The only tree search used in Monte Carlo Tree Search is when beginning from a specific root node and therefore only tree searches with these characteristics will be mentioned here. The two most common tree search algorithms are the breath-first search algorithm and the depth-first search algorithm. Breath-first search visits all of the current nodes neighbors in order and then proceeds to the neighbor’s neighbors. Depth-first search visits one of the neighbors and then proceeds to that neighbors first neighbor directly\(^8\).

In the field of Artificial Intelligence a simple tree structure for combinatorial games with two players is the Game Tree, where each depth-level either represents the first or the second player’s moves. An example of tree-searching a game tree can be to compute which probability the player has to win for the next move, and then pick that move. If an artificial player is playing a regular player, the root node will represent the current game state, and each of the root node’s children will represent a possible move for the artificial player. The artificial player simulates all possible moves (or some subset of all possible moves) and decides which move is most likely to win the game\(^9\).
2.4 Monte Carlo

A Monte Carlo algorithm is a randomized algorithm that runs in polynomial time, but might return an invalid or non-optimal result. This differs from deterministic algorithms that always returns the same correct answer, but might require super-polynomial time.

2.5 Monte Carlo Tree Search

Monte Carlo Tree Search is an algorithm for making approximately optimal decisions in artificial intelligence problems, foremost planning the next move in combinatorial games. Examples of such games are Go, Chess and Connect Four. It uses the combination of random simulations and the precision of tree search.

The MCTS algorithm iterates $X$ times. Each time four events occur:

1. Selection - Selects, from the root node, an optimal new leaf node.
2. Expansion - Expands the leaf node with a new leaf.
3. Simulation - Simulates an outcome for the combinatorial game in question.
4. Back propagation - Returns to the root node and updates the nodes it passed through in step one with new values.

With applied to Go, the AI sets the root node to the current Go board. Then $Y$ new moves are chosen at random. This is called pure random MCTS. Then each node represents a new move, either black or white. Here is an image picturing the scenario described above:

![Figure 1: Monte Carlo Search Tree](image)

The key benefit of MCTS over a regular search tree is that this MCTS uses weighted selections and randomness. This assuring that nodes which are irrelevant gets visited less often. This leads to an unbalanced tree.

Selection for each node from a parent is decided through the maximal Upper Confidence Bounds, or UCB as denoted from here on:

Selection for each node from a parent is decided through the maximal Upper Confidence Bounds, or UCB as denoted from here on:
\[ UCB_i = v_i + C \sqrt{\frac{\ln N}{n_i}}. \]

\( UCB_i \) is the value for the current node being inspected, \( v_i \) is the estimated value of the node \( i \), \( C \) is a tuned constant, \( N \) is the current path count for all nodes summed and finally \( n_i \) is the current node’s path count.

There is another approach to select the best node among a parent node’s children. This is called the Upper Confidence Bound for Trees\(^{10} \) (UCT). This is generally the more favored approach compared to UCB regarding Go computing\(^{10} \). The formula is much similar to UCB and will not be listed here.

Since every iteration requires a traversal to a leaf and then a fully simulated game, the time complexity is, somewhat approximated, only depending on the time needed to simulate a game. The complexity could be written \( O(nk) \), where \( n \) is the number of iterations and \( k \) is the average time for each game to complete.

3 Methods

As the thesis is fundamentally comparative, we have implemented a framework for AI-performance comparisons. It’s main use is to simulate large number of games between different kind of heuristics. This allows us to quickly gather a statistical basis for the results.

3.1 Framework Technical Details

The framework is written in C++ using C++11 with GCC 4.8.2, and includes a Go implementation which allows two players to play against each other.

The Go game rules implemented in the framework is simplified. Ko - situations are solved by never allowing the player to play two consecutive moves at the same board position. This was implemented since the Ko - rules are rather complex.

The board uses stone scoring which is defined by that the player who has the most stones on the board when the game has ended wins. Go has many different scoring systems but stone scoring was the easiest to implement and therefore it was used. Also the artificial players require more intelligence if another scoring system is used, i.e. area scoring or territory scoring.

3.2 Endgame

The endgame in Go is defined as when the board is mostly covered with stones.

The endgame in Go is difficult for artificial players since when stone scoring is used, the players will do irrational moves since they can still place stones at the board. To avoid strange behaviour at endgame situations each heuristic player utilizes the following endgame-heuristic:

\[ e = \text{emptyCells} - 2 \]
\[ p = \text{PlayerStones} \]
\[ o = \text{OpponentStones} \]
\[ op = \text{OpponentLastMove} \]
\[ d = p - o \]

\[ Move = \begin{cases} 
\text{Pass} & e < d \lor op = \text{Pass} \\
\text{HeuristicMove} & \text{else} 
\end{cases} \]

This allows the beneficial branches of the Monte Carlo Search Tree to win in most cases, but not all. The subtraction by two is because a board with two free cells are generally not beneficial for the player with the most stones and should in general never be achieved.

### 3.3 Heuristic Implementations

To test our thesis, the framework contains implementations of the following heuristics:

- **All-Random:** An all random heuristic.
- **Free:** A heuristic who never plays a move which gives one of it’s components a liberty degree of one.
- **Spread:** A heuristic who plays a move as far as it can from the opponent’s latest move.
- **Close:** A heuristic who plays a move as close as it can from the opponent’s latest move.
- **Mirror:** A heuristic who plays a move as in the opposing diagonal from the opponent’s latest move. If that position is not valid the heuristic uses a breadth first search to find the closest moves to the desired move.

All heuristics implement the endgame logic listed above and if each heuristic cannot place a move they pass.

### 3.4 MCTS Implementation Details

The Monte Carlo Tree Search algorithm is implemented as creating a single node with a specified heuristic for the player and it’s opponent. All MCTS players uses only their own heuristic as their own and the all-random heuristic as the opponent’s heuristic. So when each branching sequence is completed and the game simulation occurs the partial game is played between the heuristic and all-random. The MCTS node is then branched \(X\) times, and the more branches created, the better the result.
The game class handles two components, the players and the board. The board implements all Go rules and each player move has to be verified by the board before being actually played, to assert that the move is valid. The Go board is stored in a two dimensional matrix. The matrix size is defined by a template variable. Each player is represented by an interface, this lets the player implementation either be an MCTS AI player, a regular heuristic AI player or an human player. The regular heuristic AI player plays the move suggested by the heuristic.

The design diagram shows the implementation with two MCTS players. When asked for the next move, each player creates a new TreeNode, which represents the root of the tree. Then the tree expands $X$ times and finally the best move is returned.

### 3.5 Evaluation

The framework enables us to simulate a large number of games between different kind of players. Analyzing the win-ratios gives a statistical foundation to draw results from. We choose to let each pair of opponents play 1000 games against each other and noted the win-ratios. Opponents are composed in three different
ways, heuristic against heuristic, heuristic against MCTS with heuristic and
MCTS with heuristic against MCTS with heuristic. For each pair of heuristic,
the ratios in the different compositions are correlated. This correlation relates
the general performance of the heuristic with its performance impact on MCTS.

4 Results

4.1 Time Dependency Graph

As MCTS is an iterative algorithm which performance depends on the number
of visited board states.

![Win rate of pure MCTS vs. all-random](image)

Win rate of pure MCTS vs. all-random

4.2 Comparison Result Tables

Win-ratios between heuristics. The column players plays white and the row
players play the color black. White makes the first move. The ratios in each
cell describes the the column players win rate divided by the row players win
rate. They do not always add up to one since some games are tied.

<table>
<thead>
<tr>
<th>Plain/Plain</th>
<th>All-Random</th>
<th>Free</th>
<th>Mirror</th>
<th>Close</th>
<th>Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-Random</td>
<td>0.481/0.4589</td>
<td>0.4664/0.475</td>
<td>0.0675/0.7846</td>
<td>0.6287/0.3285</td>
<td>0.2413/0.6782</td>
</tr>
<tr>
<td>Free</td>
<td>0.4659/0.472</td>
<td>0.4784/0.4579</td>
<td>0.0744/0.7819</td>
<td>0.6435/0.315</td>
<td>0.2398/0.6768</td>
</tr>
<tr>
<td>Mirror</td>
<td>0.8252/0.0163</td>
<td>0.8244/0.0207</td>
<td>0.4388/0.3045</td>
<td>0.8112/0.08</td>
<td>0.5262/0.028</td>
</tr>
<tr>
<td>Close</td>
<td>0.3292/0.6323</td>
<td>0.3246/0.6349</td>
<td>0.136/0.7645</td>
<td>0.4886/0.4734</td>
<td>0.1396/0.8258</td>
</tr>
<tr>
<td>Spread</td>
<td>0.7047/0.2189</td>
<td>0.6952/0.2269</td>
<td>0.0921/0.4961</td>
<td>0.8271/0.141</td>
<td>0.4309/0.3883</td>
</tr>
</tbody>
</table>

Table 1: Heuristics without Monte Carlo tree search comparison.
The results listed above shows that the heuristics applied without MCTS has the following ranking:

1. Close
2. Free and All-Random
3. Spread
4. Mirror

<table>
<thead>
<tr>
<th>MCTS/Plain</th>
<th>All-Random</th>
<th>Free</th>
<th>Mirror</th>
<th>Close</th>
<th>Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-Random</td>
<td>0.97/0.02</td>
<td>0.96/0.02</td>
<td>0.24/0.61</td>
<td>1/0</td>
<td>0.9/0.09</td>
</tr>
<tr>
<td>Free</td>
<td>0.99/0.01</td>
<td>0.99/0.01</td>
<td>0.14/0.71</td>
<td>1/0</td>
<td>0.91/0.14</td>
</tr>
<tr>
<td>Mirror</td>
<td>0.95/0.02</td>
<td>0.97/0.01</td>
<td>0.29/0.52</td>
<td>0.98/0.01</td>
<td>0.88/0.01</td>
</tr>
<tr>
<td>Close</td>
<td>0.96/0.04</td>
<td>0.98/0.02</td>
<td>0.19/0.7</td>
<td>1/0</td>
<td>0.71/0.27</td>
</tr>
<tr>
<td>Spread</td>
<td>1/0</td>
<td>1/0</td>
<td>0.14/0.47</td>
<td>1/0</td>
<td>1/0</td>
</tr>
</tbody>
</table>

Table 2: MCTS with heuristics versus plain heuristics comparison.

The results listed above shows that the heuristics applied with MCTS has the following ranking:

1. Close
2. Free and All-Random
3. Spread
4. Mirror

<table>
<thead>
<tr>
<th>MCTS/MCTS</th>
<th>All-Random</th>
<th>Free</th>
<th>Mirror</th>
<th>Close</th>
<th>Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td>All-Random</td>
<td>0.52/0.46</td>
<td>0.67/0.33</td>
<td>0.05/0.9</td>
<td>0.81/0.19</td>
<td>0.04/0.96</td>
</tr>
<tr>
<td>Free</td>
<td>0.61/0.39</td>
<td>0.53/0.45</td>
<td>0.08/0.89</td>
<td>0.8/0.2</td>
<td>0.06/0.94</td>
</tr>
<tr>
<td>Mirror</td>
<td>0.94/0.02</td>
<td>0.92/0.01</td>
<td>0.34/0.08</td>
<td>0.99/0</td>
<td>0.82/0.03</td>
</tr>
<tr>
<td>Close</td>
<td>0.37/0.63</td>
<td>0.35/0.65</td>
<td>0.04/0.94</td>
<td>0.53/0.47</td>
<td>0/1</td>
</tr>
<tr>
<td>Spread</td>
<td>0.96/0.04</td>
<td>0.94/0.05</td>
<td>0.11/0.78</td>
<td>0.97/0.03</td>
<td>0.65/0.35</td>
</tr>
</tbody>
</table>

Table 3: MCTS with heuristic versus MCTS with heuristic.

4.3 Correlation Graph

In the following diagram $x$ is the win ratio between a pair of plain heuristics and $y$ is the win ratio for the same heuristics using MCTS.
Correlation between plain heuristics and MCTS heuristics performance, $r^2 = 0.9167$

5 Conclusion

The time dependency graph and table 2 shows some expected result of MCTS. Performance increases with running time and improves the performance compared with using the plain heuristic.

From table 1 and 3 two strong statements can be made. Firstly MCTS always improves the performance of an heuristic, as expected. Secondly the performance between a heuristics performance and its MCTS version correlates. A positive correlation between win rate in plain heuristics and MCTS would indicate that heuristics can improve performance. The above correlation graph indicates a strong positive correlation between the heuristics performance and the impact on MCTS and thus a positive answer of our thesis.

6 Discussion

6.1 Comparison Limitations

These comparisons are limited by the fact that they only involve heuristics. As stated, heuristics approaches has some well-known problems with Go against human players. It’s therefore unknown if the observed correlation would extend to MCTS versus human games. As the heuristics implements simple rules, humans are likely to learn and exploit them. The important observation from our result is therefore the general correlation between heuristics and MCTS: If someone implements a heuristic that plays well against human players, it can be used with MCTS and will then exhibit greatly improved performance.

6.2 Performance against humans

Ideally one would like to correlate the performance improvements over other heuristics with performance against human players. This is however maimed by our end-game simplification and more advanced game play exhibited by humans.
On small boards these effects are limited and our implementation thus performs well. Larger boards also exponentially increases the running time of the MCTS algorithm, leading to worse performance.

6.3 Implementation Weaknesses
The implementation has some weaknesses which should be taken into account when studying the results and the conclusion of the thesis. Foremost there are three issues which need to be emphasized, that the endgame logic of the Go implementation is rather simplified, that the scoring method used is stone scoring and finally that the Ko-rules are simplified. The endgame logic is very hard to implement correctly and as mentioned before it was decided to simplify this logic to be able to focus on other implementation-related details. The stone scoring method is used professionally but there are other scoring methods which are more common. These are however more advanced and it was decided to implement the simplest of scoring methods. The Ko-rules are in general very difficult to understand and even though these rules could be implemented in the board it was decided to use a simplified version of them. These three weaknesses result in a simplified Go implementation which could have an delusional effect on the results. If the thesis method is replicated, results could differ if the board was more elaborately implemented.

6.4 Choosing of Heuristics
It should be mentioned that there are most certainly many more successful heuristics which could be applied to the testing framework and that the thesis has only chosen a handful of the possible heuristics available. However, the heuristics chosen are rather simple and intuitive which strengthens the readers faith in the thesis results.

7 References
1. The Way To Go, Karl Baker, American Go Association
2. Martin M¨ uller, Computer Go, Artificial Intelligence 134 (2002), University of Alberta, Edmonton
3. Robin Upton. Dynamic Stochastic Control: A New Approach to Tree Search & Game-Playing, University of Warwick, UK 23 April 1998
7. Jay Burmeister and Janet Wiles, AI Techniques Used in Computer Go, Schools of Information Technology and Psychology, The University of Queensland, Australia


8 Appendix A

This appendix contains the source code of our project.

Listing 1: MCTS.hpp

```cpp
#ifndef MCTS_HPP
#define MCTS_HPP

#include <chrono>
#include <memory>
#include "move.hpp"
#include "board.hpp"
#include "treenode.hpp"
#include "playheuristic.hpp"

template<long D>
class MCTS {
    private:
        const Board<D>& board;
        const PlayHeuristic<D>& playHeuristic;
        const Color color;
    public:
        MCTS(const Board<D>& board, const PlayHeuristic<D>& playHeuristic, Color color) :
            board(board),
            playHeuristic(playHeuristic),
            color(color) {}

        Move getMove(const long timelimit) const {
            Color opponentColor = getOpponentColor(color);

            PlayHeuristicAll<D> opponentPlayHeuristic;

            auto boardCopy = board.clone();
            TreeNode<D> *treeNode = new TreeNode<D>(boardCopy,
                opponentPlayHeuristic,
                playHeuristic,
                opponentColor,
                color,
                0);

            auto start = std::chrono::steady_clock::now();
            auto end = std::chrono::steady_clock::now();
            auto ms = std::chrono::duration<double, std::milli>(end-start).count();
            do {
```
treeNode->selectAction();
end = std::chrono::steady_clock::now();
ms = std::chrono::duration<double, std::milli>
>(end-start).count();
} while (ms < timelimit);

Move move = treeNode->getMove();
delete treeNode;
return move;
};
#endif

Listing 2: MCTSAI.hpp

#ifndef MCTSAI_HPP
#define MCTSAI_HPP

#include "color.hpp"
#include "player.hpp"
#include "board.hpp"
#include "game.hpp"
#include "MCTS.hpp"
#include "playheuristic.hpp"
#include "move.hpp"

template <long D>
class MCTSAI : public Player<D> {
private:
    const Color color;
    const PlayHeuristic<D>& playHeuristic;
    const long timelimit;
public:
    MCTSAI(const Color color,
            const PlayHeuristic<D>& playHeuristic,
            const long timelimit) :
        color(color),
        playHeuristic(playHeuristic),
        timelimit(timelimit) {
    }

    virtual Move getMove(const Board<D>& board) const {
        MCTS<D> mcts(board, playHeuristic, color);
        return mcts.getMove(timelimit);
    }

    virtual const Color& getColor() const {
        return color;
    }
};
virtual void gameOver(const Board<>& board) const {
}
}
#endif

Listing 3: board.hpp

#ifndef DEFINE_BOARD
#define DEFINE_BOARD

#include <set>
#include <list>
#include <array>
#include <memory>
#include <vector>
#include <utility>
#include <iostream>
#include "move.hpp"
#include "score.hpp"
#include "color.hpp"
#include "winner.hpp"

class Point {
public:
    enum class State { Black, White, Empty };
public:
    Point() : x(0), y(0), state(State::Empty) {}
    Point(const long x, const long y, const State state) : x(x), y(y), state(state) {}
    bool operator<(const Point& rhs) const {
        return std::tie(getX(), getY(), getState()) < std::tie(rhs.getX(), rhs.getY(), rhs.getState());
    }
    virtual ~Point() {};
    State& getState() {
        return state;
    }
};
const State& getState() const {
    return state;
}

const long& getX() const {
    return x;
}

const long& getY() const {
    return y;
}

private:
    long x;
    long y;
    State state;
};

Color pointStateToColor(const Point::State& state);
Point::State colorToPointState(const Color& color);

template<long D>
class Board {
private:
public:
    using Coordinate = std::pair<long, long>;

public:
    Board(const Color color)
        : finished(false),
          latestMove(getOpponentColor(color)),
          latestKoMove(color),
          started(false),
          winner(Winner::Tie) {}

virtual ~Board() {};
virtual std::shared_ptr<Board> clone() const = 0;
virtual Point& getPoint(const long x, const long y) = 0;
virtual const Point& getPoint(const long x, const long y) const = 0;

virtual const Point::State& getPointState(const long x, const long y) const {
    return getPoint(x, y).getState();
}

virtual bool playPoint(const Color color, const long x, const long y) {
}
if (!started)
  started = true;

if (validMove(color, x, y)) {
  auto& point = getPoint(x, y);
  point.getState() = colorToPointState(color);
  updateBoard(x, y);
  latestKoMove = latestMove;
  latestMove = Move(color, point.getX(), point.
                  getY());
  return true;
} else {
  return false;
}

virtual void playPass(const Color color) {
  if (!started)
    started = true;

  latestKoMove = latestMove;
  latestMove = Move(color);
}

virtual bool validMove(const Color color, const long x, const long y) const {
  const auto& point = getPoint(x, y);
  if (point.getState() != Point::State::Empty)
    return false;
  else {
    if (!latestKoMove.isPass()) {
      const auto& coord = latestKoMove.getMove();
      if (std::get<0>(coord) == x && std::get
          <1>(coord) == y) {
        return false;
      }
    }
  }

  for (const auto& coord : getAdjacent(x, y)) {
    const auto& adjacent = getPoint(coord.
        first, coord.second);
    if (adjacent.getState() == Point::State::Empty)
      return true;
  }

  for (const auto& coord : getAdjacent(x, y)) {
    const auto& adjacent = getPoint(coord.
        first, coord.second);
if (pointStateToColor(adjacent.getState())
        == color) {
    std::vector<Point> component;
    findComponent(adjacent.getX(),
                  adjacent.getY(),
                  adjacent.getState(),
                  component);
    std::set<Point> free;
    getFreedom(component, free);
    if (free.size() > 1) {
        return true;
    } else {
    std::vector<Point> component;
    findComponent(adjacent.getX(),
                  adjacent.getY(),
                  adjacent.getState(),
                  component);
    std::set<Point> free;
    getFreedom(component, free);
    if (free.size() <= 1) {
        return true;
    }
}
}
return false;

virtual void gameOver() {
    finished = true;
    const auto score = getScore();
    if (score.getBlackScore() > score.getWhiteScore())
        winner = Winner::Black;
    else if (score.getBlackScore() < score.getWhiteScore())
        winner = Winner::White;
    else
        winner = Winner::Tie;
}

virtual bool isStarted() const {
    return started;
}

virtual bool isFinished() const {
    return finished;
}
virtual long getCountBlack() const {
    return getCountState(Point::State::Black);
}

virtual long getCountEmpty() const {
    return getCountState(Point::State::Empty);
}

virtual long getCountWhite() const {
    return getCountState(Point::State::White);
}

virtual Winner getWinner() const {
    if (!isFinished())
        throw std::logic_error("Game is not over");
    else
        return winner;
}

virtual Score getScore() const {
    return areaScore();
}

virtual Move getLatestMove() const {
    if (!isStarted())
        throw std::logic_error("Game not started");
    else
        return latestMove;
}

virtual int getLibertiesForComponent(const long x, const long y, const Color color) const {

    std::vector<Point> component;
    Point::State state = color == Color::White ?
        Point::State::White : Point::State::Black;
    findComponent(x, y, state, component);
    if (component.empty()) {
        return D;
    }

    std::set<Point> free;
    getFreedom(component, free);
    return free.size();
}
protected:

    virtual void updateBoard(const long x, const long y)
    {
        const auto& point = getPoint(x, y);
        Color color = pointStateToColor(point.getState());

        auto adjacent = getAdjacent(x, y);
        for (const auto& coord : adjacent)
            {
                const auto& point = getPoint(coord.first, coord.second);
                if (point.getState() == colorToPointState(getLatestMove().getColor()))
                    updateComponent(coord.first, coord.second);
            }

        for (const auto& coord : adjacent)
            {
                const auto& point = getPoint(coord.first, coord.second);
                if (point.getState() != Point::State::Empty)
                    updateComponent(coord.first, coord.second);
            }
    }

virtual long getCountState(const Point::State state) const {
    long count = 0;
    for (long y = 0; y < D; ++y)
        {
            for (long x = 0; x < D; ++x)
                {
                    if (getPoint(x, y).getState() == state)
                        ++count;
                }
        }
    return count;
}

virtual Score areaScore() const {
    Score score:
    for (long y = 0; y < D; ++y)
        {
            for (long x = 0; x < D; ++x)
                {
                    const auto& point = getPoint(x, y);
                    if (point.getState() == Point::State::White)
                        ++score.getWhiteScore();
                }
        }
    return score;
}
else if (point.getState() == Point::State::Black)
    ++score.getBlackScore();
}
}
return score;
}

private:
void updateComponent(const long x, const long y) {
    const auto& point = getPoint(x, y);
    std::vector<Point> component;
    if (point.getState() != Point::State::Empty)
        findComponent(x, y, point.getState(), component);
    std::set<Point> free;
    getFreedom(component, free);
    if (free.empty()) {
        for (const auto& point : component) {
            auto& point = getPoint(point.getX(), point.getY()).getState() = Point::State::Empty;
        }
    }
}

void getFreedom(const std::vector<Point>& component, std::set<Point>& free) const {
    for (const auto& point : component) {
        for (const auto& adjacent : getAdjacent(point.getX(), point.getY())) {
            auto& point = getPoint(adjacent.first, adjacent.second);
            if (point.getState() == Point::State::Empty)
                free.insert(point);
        }
    }
}

void findComponent(const long x, const long y, const Point::State state,
                    const std::vector<Point>& component) const {

std::array<std::array<bool, D>, D> visited;

for (long y = 0; y < D; ++y)
    for (long x = 0; x < D; ++x)
        visited[y][x] = false;

return findComponent(x, y, state, visited, component);
}

static std::vector<Coordinate> getAdjacent(const long x, const long y) {
    std::vector<Coordinate> adjacent;
    if (x+1 < D)
        adjacent.emplace_back(std::make_pair(x + 1, y));
    if (x-1 >= 0)
        adjacent.emplace_back(std::make_pair(x - 1, y));
    if (y+1 < D)
        adjacent.emplace_back(std::make_pair(x, y + 1));
    if (y-1 >= 0)
        adjacent.emplace_back(std::make_pair(x, y-1));

    return adjacent;
}

private:

void findComponent(const long x, const long y, const Point::State state, 
    std::array<std::array<bool, D>, D>& visited, 
    std::vector<Point>& component) 
    const 
    if (visited[y][x]) {
        return;
    } else {
        visited[y][x] = true;

        const auto& point = getPoint(x, y);
        if (point.getState() == state) {
            component.emplace_back(point);
            for (const auto& coord : getAdjacent(x, y)) 
                findComponent(coord.first, coord.second, state, 
                               visited, component);
        }
bool started;
bool finished;
Move latestMove;
Move latestKoMove;
Winner winner;
}

Color pointStateToColor(const Point::State& state) {
    return state == Point::State::White ? Color::White :
            Color::Black;
}

Point::State colorToPointState(const Color& color) {
    return color == Color::White ? Point::State::White :
            Point::State::Black;
}

#endif

Listing 4: color.hpp

#elif defined DEFINE_COLOR
#define DEFINE_COLOR

#include <iostream>

eenum class Color { Black, White }; 

Color getOpponentColor(const Color color) {
    return color == Color::White ? Color::Black : Color::White ;
}

std::ostream& operator << (std::ostream& os, const Color& color)
{
    os << (color == Color::Black ? "black" : "white");
    return os;
}

#endif

Listing 5: endgame.hpp

#elif defined DEFINE_ENDGAME
#define DEFINE_ENDGAME

24
#include "game.hpp"

template<long D>
class EndGame : public Game<D> {
public:
    EndGame(const Player<D>& first, const Player<D>& second)
        : Game<D>(first, second) {
    }

    virtual void start(Board<D>&& board) {
        bool firstPassed = false;
        bool secondPassed = false;

        auto& first = Game<D>::getFirstPlayer();
        auto& second = Game<D>::getSecondPlayer();
        while (!firstPassed || !secondPassed) {
            hookTurn(board);
            if (!firstPassed) {
                const auto move = first.getMove(board);
                if (!move.isPass()) {
                    const auto& coord = move.getMove();
                    if (!board.playPoint(first.getColor(),
                        std::get<0>(coord),
                        std::get<1>(coord))) {
                        throw std::logic_error("First
                            player made an invalid move");
                    }
                    hookMove(board);
                } else {
                    hookPass(board);
                    firstPassed = true;
                }
            } else {
            }
        }
    }

}
```cpp
const auto& coord = move.getMove();
if (!board.playPoint(second.getColor(), std::get<0>(coord), std::get<1>(coord)))
{
    throw std::logic_error("Second player made an invalid move");
}
hookMove(board);
else {
    hookPass(board);
    secondPassed = true;
}
}
}
}
}
gameOver(board);

#endif
Listing 6: game.hpp

#ifndef DEFINE/Game
#define DEFINE/Game

#include "board.hpp"
#include "player.hpp"

template<

```
const auto& coord = move.getMove();
if (!board.playPoint(first.getColor(),
    std::get<0>(
        coord),
    std::get<1>(
        coord))) {
    throw std::logic_error("First player made an invalid move");
}
hookMove(board);
} else {
    board.playPass(first.getColor());
    if (pass) {
        break;
    }
    hookPass(board);
    pass = true;
}

const auto move = second.getMove(board);
if (!move.isPass()) {
    pass = false;
    const auto& coord = move.getMove();
    if (!board.playPoint(second.getColor(),
        std::get<0>(
            coord),
        std::get<1>(
            coord))) {
        throw std::logic_error("Second player made an invalid move");
    }
    hookMove(board);
} else {
    board.playPass(second.getColor());
    if (pass) {
        break;
    }
    hookPass(board);
    pass = true;
}
}
gameOver(board);
virtual const Player<D>& getFirstPlayer() {  
    return first;
}

template<>
virtual const Player<D>& getSecondPlayer() { 
    return second;
}

virtual void gameOver(Board<D>& board) { 
    board.gameOver();
    first.gameOver(board);
    second.gameOver(board);
    hookGameOver(board);
}

virtual void hookTurn(const Board<D>& board) const { 
}

virtual void hookMove(const Board<D>& board) const { 
}

virtual void hookPass(const Board<D>& board) const { 
}

virtual void hookGameOver(const Board<D>& board) const { 
}

private:
    const Player<D>& first;
    const Player<D>& second;
};

Listing 7: heuristicplayer.hpp

#define HEURISTIC_PLAYER

#include <tuple>
#include <algorithm>
#include <iterator>
#include <random>
#include <chrono>
```cpp
#include "player.hpp"
#include "playheuristic.hpp"
#include "random.hpp"

template <long D>
class HeuristicPlayer : public Player<D> {
private:
    const Color color;
    const PlayHeuristic<D>& playHeuristic;

public:
    HeuristicPlayer(const Color color, const PlayHeuristic<D>& playHeuristic) :
        color(color), playHeuristic(playHeuristic) { }

    virtual Move getMove(const Board<D>& board) const {
        auto moves = playHeuristic.getMoves(board, color);
        std::shuffle(moves.begin(), moves.end(), generator);
        auto& move = moves[0];
        long x = std::get<0>(move);
        long y = std::get<1>(move);
        return (x == -1 && y == -1 ? Move(color) : Move(
            color, x, y));
    }

    virtual const Color& getColor() const {
        return color;
    }

    virtual void gameOver(const Board<D>& board) const {
        }
};
```

Listing 8: main.cpp

```cpp
#include "text/game.hpp"
#include "text/player.hpp"
#include "matrixboard.hpp"
#include "heuristicplayer.hpp"
#include "playheuristicall.hpp"
#include "playheuristicclose.hpp"
#include "playheuristicspread.hpp"
#include "playheuristicmirror.hpp"
#include "playheuristicfreedom.hpp"
```
```cpp
#include "MCTSAI.hpp"

static const long D = 6;
static const long rounds = 100;

int main() {
    PlayHeuristicAll<D> randomHeuristic;
    PlayHeuristicFreedom<D> freeHeuristic;
    PlayHeuristicMirror<D> mirrorHeuristic;
    PlayHeuristicClose<D> closeHeuristic;
    PlayHeuristicSpread<D> spreadHeuristic;

    std::array<PlayHeuristic<D>*, 5> heuristics =
        { &randomHeuristic, &freeHeuristic, &mirrorHeuristic, &closeHeuristic, &spreadHeuristic };

    text::Player<D> fabian(Color::White);
    MCTSAI<D> ai(Color::Black, closeHeuristic, 1000);
    MatrixBoard<D> board(fabian.getColor());
    Game<D> game(fabian, ai);
    game.start(board);

    /*long i = 0;
    for (const auto& white : heuristics) {
        long j = 0;
        for (const auto& black : heuristics) {
            MCTSAI<D> blackPlayer(Color::Black, *black, 64);
            MCTSAI<D> whitePlayer(Color::White, *white, 64);
            Game<D> game(whitePlayer, blackPlayer);

            long blackWins = 0;
            long whiteWins = 0;

            try {
                for (long i = 0; i < rounds; ++i) {
                    MatrixBoard<D> board(whitePlayer.getColor());
                    game.start(board);
                    if (board.getWinner() == Winner::Black)
                        ++blackWins;
                    else if (board.getWinner() == Winner::White)
```
```cpp
++whiteWins;

} catch (const char* error) {
    std::cout << "Error: " << error << std::endl;
}

std::cout << i << " vs " << j << ": "
    << whiteWins / (double) rounds << "/
    << blackWins / (double) rounds << std::endl;

++j;
} ++i;
}*/

Listing 9: matrixboard.hpp

#ifndef DEFINE_MATRIXBOARD
#define DEFINE_MATRIXBOARD

#include "board.hpp"

template<long D>
class MatrixBoard : public Board<D> {
public:
    MatrixBoard(const Color color) : Board<D>(color),
    matrix() {
        for (long y = 0; y < D; ++y)
            for (long x = 0; x < D; ++x)
                matrix[y][x] = Point(x, y, Point::State::Empty);
    }
    virtual ~MatrixBoard() {};

    virtual std::shared_ptr<Board<D>> clone() const {
        return std::shared_ptr<MatrixBoard<D>>(new MatrixBoard(*this));
    }

    virtual Point& getPoint(const long x, const long y) {
        if (x < 0 || y < 0 || x >= D || y >= D)
            throw std::out_of_range("func.");
        return matrix[y][x];
    }
};

#endif
```

virtual const Point& getPoint(const long x, const long y) const {
    if (x < 0 || y < 0 || x >= D || y >= D)
        throw std::out_of_range("func");
    return matrix[y][x];
}

private:
    Point matrix[D][D];
};

Listing 10: move.hpp

#ifndef DEFINE
#define DEFINE
#include <tuple>
#include "color.hpp"

class Move {
public:
    using Coordinate = std::tuple<long, long>;

    Move(const Color color) :
        pass(true), color(color), coordinate(std::make_tuple(0, 0)) {}

    Move(const Color color, const long x, const long y) :
        pass(false), color(color), coordinate(std::make_tuple(x, y)) {}

    bool isPass() const {
        return pass;
    }

    Color getColor() const {
        return color;
    }

    Coordinate getMove() const {
        if (pass)
            throw "Player passed";
        else
            return coordinate;
    }
}
private:
    bool pass;
    Color color;
    Coordinate coordinate;
};
#endif

Listing 11: player.hpp

#ifndef PLAYER_HPP
#define PLAYER_HPP

#include <tuple>
#include "move.hpp"
#include "board.hpp"

template <long D>
class Player {
public:
    virtual ~Player() {} 
    virtual const Color& getColor() const = 0;
    virtual Move getMove(const Board<D>& board) const = 0;
    virtual void gameOver(const Board<D>& board) const = 0;
};
#endif

Listing 12: playheuristic.hpp

#ifndef PLAY_HEURISTIC_HPP
#define PLAY_HEURISTIC_HPP

#include <tuple>
#include <vector>
#include <cmath>
#include "board.hpp"
#include "color.hpp"

template <long D>
class PlayHeuristic {
public:
    virtual ~PlayHeuristic() {} 
    virtual std::vector<std::tuple<long, long>>
    getMoves(const Board<D>& board, const Color color)
    const = 0;
    static double getDistance(std::tuple<long, long>&
         first, std::tuple<long, long>& second) {

long xDiff = std::get<0>(first) - std::get<0>(second);
long yDiff = std::get<1>(first) - std::get<1>(second);
return sqrt(xDiff * xDiff + yDiff * yDiff);
};
#endif

Listing 13: playheuristicall.hpp

#ifndef PLAY_HEURISTIC_ALL_HPP
#define PLAY_HEURISTIC_ALL_HPP

#include <vector>
#include <tuple>
#include <cmath>
#include "playheuristic.hpp"
#include "board.hpp"

template <long D>
class PlayHeuristicAll : public PlayHeuristic<D> {
public:

virtual std::vector<std::tuple<long, long>> getMoves(const Board<D>& board, const Color color) const {

int e = board.getCountEmpty() - 2;
int b = board.getCountBlack();
int w = board.getCountWhite();

int my = color == Color::White ? w : b;
int opponent = my == w ? b : w;

int diff = my - opponent;

std::vector<std::tuple<long, long>> moves;

if (e < diff) {
    moves.push_back(std::make_tuple(-1, -1));
} else if (diff > 0 && board.getLatestMove().isPass()) {
    moves.push_back(std::make_tuple(-1, -1));
} else {
    for (long x = 0; x < D; ++x) {
        for (long y = 0; y < D; ++y) {
            if (board.validMove(color, x, y)) {
                moves.push_back(std::make_tuple(x, y));
            }
        }
    }
}
if (moves.size() == 0) {
    moves.push_back(std::make_tuple(-1, -1));
}

return moves;
};
#endif

Listing 14: playheuristicclose.hpp

#ifndef PLAY_HEURISTIC_CLOSE_HPP
#define PLAY_HEURISTIC_CLOSE_HPP

#include <vector>
#include <tuple>
#include <cmath>
#include <algorithm>
#include "playheuristic.hpp"
#include "playheuristicdistance.hpp"
#include "board.hpp"

template <long D>
class PlayHeuristicClose : public PlayHeuristicDistance<
    > {
public:
    virtual std::vector<std::tuple<long, long>> >
        getMoves(const Board<D> &board, const Color color) const {
            return PlayHeuristicDistance<D>::getMoves(board, color, false);
        }
};
#endif

Listing 15: playheuristicdistance.hpp

#ifndef PLAY_HEURISTIC_DISTANCE_HPP
#define PLAY_HEURISTIC_DISTANCE_HPP

#include <vector>
#include <tuple>
#include <cmath>
#include <algorithm>
```cpp
#include "playheuristic.hpp"
#include "board.hpp"

template <long D>
class PlayHeuristicDistance : public PlayHeuristic<D> {
public:
    static const int MAGIC = 5;

    virtual std::vector<std::tuple<long, long>> getMoves(const Board<D>& board, const Color color) const = 0;
    virtual std::vector<std::tuple<long, long>> getMoves(const Board<D>& board, const Color color, bool spread) const {
        int e = board.getCountEmpty() - 2;
        int b = board.getCountBlack();
        int w = board.getCountWhite();

        int my = color == Color::White ? w : b;
        int opponent = my == w ? b : w;

        int diff = my - opponent;

        std::vector<std::tuple<long, long>> moves;

        if (e < diff) {
            moves.push_back(std::make_tuple(-1, -1));
        } else if (!board.isStarted() || board.getLatestMove().isPass()) {
            for (long x = 0; x < D; ++x) {
                for (long y = 0; y < D; ++y) {
                    if (board.validMove(color, x, y)) {
                        moves.push_back(std::make_tuple(x, y));
                    }
                }
            }
        } else if (diff > 0 && board.getLatestMove().isPass()) {
            moves.push_back(std::make_tuple(-1, -1));
        } else {
            if (board.getLatestMove().isPass() || !board.is Started()) {
                for (long x = 0; x < D; ++x) {
                    for (long y = 0; y < D; ++y) {
                        if (board.validMove(color, x, y)) {
                            moves.push_back(std::make_tuple(x, y));
                        }
                    }
                }
            }
        }
    }
};
```
```cpp
} }
} else {
    auto coords = board.getLatestMove().getMove();
    std::vector<std::tuple<double, std::tuple<long, long>>> possibleMoves;

    for (long x = 0; x < D; ++x) {
        for (long y = 0; y < D; ++y) {
            if (board.validMove(color, x, y)) {
                auto move = std::make_tuple(x, y);
                possibleMoves.push_back(std::make_tuple(PlayHeuristic<D>::getDistance(coords, move), move));
            }
        }
    }

    std::sort(std::begin(possibleMoves), std::end(possibleMoves),
               [&spread](const std::tuple<double, std::tuple<long, long>>& S, const std::tuple<double, std::tuple<long, long>>& R) {
                   if (spread) {
                       return std::get<0>(S) > std::get<0>(R);
                   } else {
                       return std::get<0>(S) < std::get<0>(R);
                   }
               });

    std::vector<std::tuple<long, long>> candidates;
    for (long i = 0; i < MAGIC && i < possibleMoves.size(); ++i) {
        moves.push_back(std::get<1>(possibleMoves[i]));
    }
```
if (moves.size() == 0) {
    moves.push_back(std::make_tuple(-1, -1));
}

return moves;
};

Listing 16: playheuristicfreedom.hpp

#ifndef PLAY_HEURISTIC_FREEDOM_HPP
#define PLAY_HEURISTIC_FREEDOM_HPP

#include <vector>
#include <tuple>
#include <cmath>
#include "playheuristic.hpp"
#include "board.hpp"

template <long D>
class PlayHeuristicFreedom : public PlayHeuristic<D> {
public:

    virtual std::vector<std::tuple<long, long>> > getMoves(const Board<D>& board, const Color color) const {

        int e = board.getCountEmpty() - 2;
        int b = board.getCountBlack();
        int w = board.getCountWhite();

        int my = color == Color::White ? w : b;
        int opponent = my == w ? b : w;

        int diff = my - opponent;

        std::vector<std::tuple<long, long>> > moves;

        if (e < diff) {
            moves.push_back(std::make_tuple(-1, -1));
        } else if (diff > 0 && board.getLatestMove().isPass()) {
            moves.push_back(std::make_tuple(-1, -1));
        } else {
            for (long x = 0; x < D; ++x) {
                for (long y = 0; y < D; ++y) {
                    if (board.validMove(color, x, y)
```cpp
&& board.getLibertiesForComponent(x, y, color) > 1) {
    moves.push_back(std::make_tuple(x, y));
}
}

if (moves.size() == 0) {
    moves.push_back(std::make_tuple(-1, -1));
}

return moves;
}
};
#endif

Listing 17: playheuristicmirror.hpp

#ifndef PLAY_HEURISTIC_MIRROR_HPP
#define PLAY_HEURISTIC_MIRROR_HPP

#include <vector>
#include <tuple>
#include <cmath>
#include <array>
#include <queue>
#include "playheuristic.hpp"
#include "board.hpp"

template <long D>
class PlayHeuristicMirror : public PlayHeuristic<D> {
public:

    virtual std::vector<std::tuple<long, long>> getMoves(const Board<D>& board, const Color color) const {

        int e = board.getCountEmpty() - 2;
        int b = board.getCountBlack();
        int w = board.getCountWhite();

        int my = color == Color::White ? w : b;
        int opponent = my == w ? b : w;

        int diff = my - opponent;

        std::vector<std::tuple<long, long>> moves;
```
if (e < diff) {
    moves.push_back(std::make_tuple(-1, -1));
} else if (!board.isStarted() || board.getLatestMove().isPass()) {
    for (long x = 0; x < D; ++x) {
        for (long y = 0; y < D; ++y) {
            if (board.validMove(color, x, y)) {
                moves.push_back(std::make_tuple(x, y));
            }
        }
    }
} else if (diff > 0 && board.getLatestMove().isPass()) {
    moves.push_back(std::make_tuple(-1, -1));
} else {
    auto coords = board.getLatestMove().getMove();
    std::queue<std::tuple<long, long>> queue;
    long mirrorX = D - std::get<0>(coords) - 1;
    long mirrorY = D - std::get<1>(coords) - 1;
    queue.push(std::make_tuple(mirrorX, mirrorY));
    std::array<bool, D * D> visited;

    while (!queue.empty()) {
        auto& current = queue.front();
        queue.pop();
        long x = std::get<0>(current);
        long y = std::get<1>(current);

        if (board.validMove(color, x, y)) {
            moves.push_back(std::make_tuple(x, y));
        } else if (moves.empty()) {
            if (x != 0 && !visited[y * D + (x - 1)]) {
                queue.push(std::make_tuple(x - 1, y));
                visited[y * D + (x - 1)] = true;
            }
        }
        if (x != D - 1 && !visited[y * D + (x + 1)]) {
            queue.push(std::make_tuple(x + 1, y));
        }
    }
}
visited[y * D + (x + 1)] = true;
}
if (y != 0 && !visited[(y - 1) * D + x]) {
    queue.push(std::make_tuple(x, y - 1));
    visited[(y - 1) * D + x] = true;
}
if (y != D-1 && !visited[(y + 1) * D + x]) {
    queue.push(std::make_tuple(x, y + 1));
    visited[(y + 1) * D + x] = true;
}
}
}
if (moves.size() == 0) {
    moves.push_back(std::make_tuple(-1, -1));
}
return moves;
};
#endif

Listing 18: playheuristicspread.hpp

#ifndef PLAY_HEURISTIC_SPREAD_HPP
#define PLAY_HEURISTIC_SPREAD_HPP
#include <vector>
#include <tuple>
#include <cmath>
#include <algorithm>
#include "playheuristic.hpp"
#include "playheuristicdistance.hpp"
#include "board.hpp"

template <long D>
class PlayHeuristicSpread : public PlayHeuristicDistance< D> {
public:
    virtual std::vector<std::tuple<long, long>> >
    getMoves(const Board<D>& board, const Color color) const {

```cpp
return PlayHeuristicDistance<D>::getMoves(board, color, true);
}
};
#endif

Listing 19: random.hpp

#ifndef RANDOM_HPP
#define RANDOM_HPP

#include <random>

static std::random_device rd;
static std::mt19937 generator(rd());
#endif

Listing 20: score.hpp

#ifndef DEFINE_SCORE
#define DEFINE_SCORE

class Score {
public:
  Score() : blackScore(0), whiteScore(0) {
  }

  Score(const long blackScore, const long whiteScore) :
    blackScore(blackScore), whiteScore(whiteScore) {
  }

  virtual long& getBlackScore() {
    return blackScore;
  }

  virtual const long& getBlackScore() const {
    return blackScore;
  }

  virtual long& getWhiteScore() {
    return whiteScore;
  }

  virtual const long& getWhiteScore() const {
    return whiteScore;
  }

42
```
private:
    long blackScore;
    long whiteScore;
};
#endif