Harry Kay - 19387837

```cpp
/*
* For every possible move for the current board state, find the move that has
* the highest evaluation from evaluate(). Has similar implementation to
* maxValue(), however, it also has book keeping to record which move had the
* highest evaluation.
*
* Computational complexity analysis:
*
*             Let moves.size() = n
*             Let depth_ = m
*             Complexity: O(n^m)
*
* Explanation:
* minimax() calls minValue which is indirectly recursive with maxValue()
* for a total of m times. For each of these calls, moves are evaluated for
* a total of n times. For example, if we assume that n = 3, and m = 2, then
* we have 3 moves for every level of depth, for a total of 6 moves that
* need to be processed:
*
* m_0: n_0, n_1, n_2
* m_1: n_0, n_1, n_2
*
* which is 3^2 = 6, or n^m
* Therefor, the computational complexity is O(n^m)
*
* Pre-condition:
* - The board is not full.
*
* Post-condition:
* - The board is left unchanged from simulations.
*
* @return A valid move.
*/

Move HardComputer::minimax()
{
	map<float_t, Move> valueMoveMap;
	auto moves = gameAnalyzer_->findAllValidMoves(playerColor_);

	// For each move, record its evaluation.
	for (const auto& move : moves)
	{
		board_->setMove(move);
		auto value = minValue(depth_ - 1);
		valueMoveMap[value] = move;
		board_->undoMove();
	}

	// Return the move that had the highest evaluation.
	return valueMoveMap.rbegin()->second;
}
```

```cpp
float_t HardComputer::maxValue(const uint32_t depth)
{
	// Base case: board is full or depth of search reaches 0.
	if (depth == 0 || board_->isBoardFull())
	{
		return evaluate(OCCUPANT_COUNT_DIFFERENCE);
	}

	// general case: there are board states to search.
	auto value = -INFINITY;
	auto moves = gameAnalyzer_->findAllValidMoves(playerColor_);
	for (const auto& move : moves)
	{
		board_->setMove(move);
		value = max(value, minValue(depth - 1));
		board_->undoMove();
	}
	return value;
}

float_t HardComputer::minValue(const uint32_t depth)
{
	// Base case: board is full or depth of search reaches 0.
	if (depth == 0 || board_->isBoardFull())
	{
		return evaluate(OCCUPANT_COUNT_DIFFERENCE);
	}

	// general case: there are board states to search.
	auto value = INFINITY;
	auto moves = gameAnalyzer_->findAllValidMoves(oppositionColor_);
	for (const auto& move : moves)
	{
		board_->setMove(move);
		value = min(value, maxValue(depth - 1));
		board_->undoMove();
	}
	return value;
}
```

```cpp
float_t HardComputer::evaluate(const HeuristicMethod method) const
{
	const auto whiteCount
		= static_cast<float_t>(gameAnalyzer_->countCellsWithColor(WHITE));
	const auto blackCount
		= static_cast<float_t>(gameAnalyzer_->countCellsWithColor(BLACK));
	const auto winner = gameAnalyzer_->findWinnersColor();

	switch (method)
	{
	case OCCUPANT_COUNT_DIFFERENCE:

		// Counts the number of occupancies and returns a difference that is
		// maximized for the maxing player and minimized for the mining player.
		if (playerColor_ == WHITE)
		{
			return whiteCount - blackCount;
		}
		return blackCount - whiteCount;

	case GAME_RESULT_ENCODING:

		//Encodes the result of games with a 1 for a win or a - 1 for a loss.
		if (winner == playerColor_)
		{
			return 1.0;
		}
		if (winner == oppositionColor_)
		{
			return -1.0;
		}
	}
	return 0.0;
}
```