Assignment #1
Black & White

by Adam Hooper, 260055737

Submitted to Yon Visell
Thursday, February 2nd, 2006
McGill University

Introduction

This report skims over most issues deliberated while coding a computerized agent for the Black & White board game. It aims to delve from higher-level to lower-level, only specifying details when the algorithm is not obvious. This report is intended to be read with the (commented) header files nearby for reference.

Object Design

At the highest level, `main.cpp` initializes a `State` and two `Agents`. It asks the `Agents` for five piece positions each (in turn); then it asks each agent in turn to provide a move. Each time an `Agent` provides a response, the `State` is updated and then is fed back into both `Agents`. (`Agents` thus see every `State`, even if they won't be asked for a move from every `State`; they may use the extra information to track the progress of the entire game.)

A `State` consists of a `Board` layout and the current player.

The `HumanAgent` specialization of `Agent` gets its moves from a human player, through standard input. The `ComputerAgent` specialization of `Agent` finds a move using an iterative-deepening minimax algorithm, discussed below. It is passed an `Evaluator` template parameter. For each available move, it generates a search `Node` (based on the current `State` and the `Agent`'s color) and finds its value (using `Evaluator`). It returns the move which created the `Node` with the highest value.

Implementation

The program was coded using object-oriented C++. In general, pointers were avoided (besides the `Agent` specializations, there is no inheritance and so no need for pointers).

Many objects are completely uninteresting in the context of this report. The header and implementation files will serve as their only documentation. These are `Agent`, `AgentFactory`, `Coord`, `HumanAgent`, `List`, `Move`, `Piece`, `State`, `global.h`, and `main.cpp`.

Board

A `Board` is represented in RAM using two 32-bit integers (bitboards). The first, `hasPiece`, stores a `1` for each board position which has a piece. The second, `pieceIsWhite`, stores a `1` for each of those pieces which is white. Clearly, `hasPiece & pieceIsWhite` returns a bitboard of all white pieces; `hasPiece & ~pieceIsWhite` returns a bitboard of all black pieces.

One of the most heavily-used functions of `Board` is its `winner` function: a check for a terminal state. It simply can't be avoided: every single `Node` must check its `Board` for terminal state. This function was optimized to a great extent, using tables.

There are two main steps to performing this check. The first is to check for 5-in-a-row sets of pieces of like color (a standard win). The algorithm – a simple table scan – will be described shortly. Next, if there is no win, the board must be checked to see whether the current player can make a move. We can optimize away some of these checks: if the current player has more than three pieces he can always make a move; if he has exactly three pieces we can check the bitboards for the four possible board positions which checkmate him; otherwise, we have to search for any move. This means searches for moves must be optimized.

There are 140 total moves available (captures and regular). Each move consists of:

- one board position containing a piece of the current player's color;
- zero or one board position containing a piece of the opponent's color; and
- one board position which is empty.

A very fast way to search for moves is to simply build a table with all these moves, with trivial bitboards. Searching for a move boils down to three `&` operations for each of 140 table entries. `Board::getNextMoves` does this, returning a list of moves found in the table scan.

This algorithm is also used to check a board's terminal state. The twelve terminal states are first checked using a similar table scan algorithm, and then a search for a move is performed. As soon as a single move is found (most of the time this will take *much* less than 140 tests), we return that the board is not in a terminal state.

The moves and board positions are found in `winning-bitsets.h`, which was generated by a Python script. The same pattern is also used in some evaluators, described later in this document.

ComputerAgent

The `ComputerAgent` class is a relatively straightforward implementation of iterative-deepening minimax with alpha-beta pruning. If more than one optimal node is found, one is randomly picked.

The solution to our seven-second deadline is simple and pragmatic, though not optimal. The agent checks after each iteration whether to stop. Assuming that the average fanout is around 9. we stop if our iteration ended after 750ms total have elapsed. This would imply that another iteration would probably cost too much. This is not optimal: in fact, the optimizations to Board put this program right on the brink between seven- and eight-ply operation. If a mechanism had been used which could cancel a running iteration, it could have gained some more eight-ply computations.

The opening game is actually very different from the rest of play. The fanout is far bigger, and the actual jumps are ignored because of the horizon effect (that is, the jumps will only be performed after the opening game, but those are beyond the search depth, so the agent may place pieces in vulnerable positions).

The first attempt at solving this problem was to have the search pretend the board has ten pieces, and have `Node` *only* apply *moves* and never add pieces. This had the advantage of avoiding placements which would lead to obvious capture. But there was a major flaw: this algorithm could never detect and avert a win in the opening game.

The final product is conceptually simpler (though it takes more code): we simply search the tree as we should, accepting the disadvantages of the horizon effect and low depth. The same heuristic function is used for incomplete boards. (The heuristic function *could* check whether the board is complete and treat it differently if it isn't; none of the implemented functions do this, though.) In the end, the search depth will consistently reach six-ply. The final heuristic checks for partial lines, which gives a defensive advantage against the opponent's lines and offensively helps create lines. But the biggest advantage is a kind of loophole: the last two or three moves of the opening game, the search tree will delve far enough to see captures; usually this will be enough to patch up the relatively random strategy used in the first seven moves with three decent moves near the end.

Optimizations

Within the minimax algorithm, if a max node ever drops below -0.5 the min node will simply return its

value. -0.5 is a “magic number”: below it we're at a big disadvantage and we can't hope to win, anyway. We aren't interested in searching losing nodes.

This optimization actually presents a trade-off. Instead of only applying the heuristic function at terminal nodes, we apply it at every node. If the heuristic function's cost is low enough, and enough search branches are pruned, we end up saving time.

Heuristic Functions

One heuristic function was developed for question 1: `Q1Evaluator`. It simply returns 0. (The heuristic is only called if the board is not in a terminal state, so this satisfies the requirements of Question 1.)

For Question 2, three heuristic functions were created, each slightly more complex than the one before:

- `Q2Evaluator`: assigns a higher value for more pieces.
- `Q2Evaluator2`: same as `Q2Evaluator`, but a sequence of four pieces in a line is given a small bonus (less than that of a piece). This is to encourage the agent to aim for its goal, even when it can't see far enough ahead.
- `Q2Evaluator3`: same as `Q2Evaluator2`, but if four pieces aren't in a line, an even smaller bonus is given for three pieces in a line, for up to three such lines. (The algorithm would slightly break down if *four* such lines were found, but this would imply that we have at least 8 pieces, in which case a win is probably nearby anyway, so we don't care.)

Each heuristic function was run against the function before it. The results were definitive: the functions above are listed from worst to best.

Failed Optimizations

During the course of development, many other attempts were made to reach 8 ply more reliably while not losing any search branches. These failed. The `Hash` implementation was submitted with the code as an artifact to potentially wean partial bonus marks for features which ended up not working and so were removed.

- Caching board positions which were actually played, and their moves. This optimization worked, but it often led to a draw. It was removed so that `ComputerAgent` could randomize its return value. Removing the cache prevented draws in the tournament.
- Caching all values of board positions greater than 5-ply. When nodes are later found with the same position, at the same depth or lower, the cached value is used. One would expect this to provide a large advantage: most moves can be undone, so intuitively it seems like we'd be reducing the fanout by around 1. No speedup was observed.

Results

In the tournament, the `ComputerAgent<Q2Evaluator3>` fared very well. In its first game it slowly but surely developed a significant advantage against its opponent, but the 50-turn limit was reached, producing a draw. Its three next opponents posed no challenge: it won against each in under 30 moves.

And how does it fare against human players? Except for an isolated win, I have not been able to defeat my `ComputerAgent<Q2Evaluator3>`. I've created a monster!