ECSE 526B
Artificial Intelligence

Instructor: Yon Visell

Assignment #1: **Black & White**

by

Samuel Audet
saudet@cim.mcgill.ca
ID: 260184380

January 30, 2006
(Revised May 3, 2006)

Electrical and Computer Engineering Department
3480 University Street Room 633
Montréal, Québec,
Canada H3A 2K6

Introduction

For this assignment, I programmed a game-playing agent for playing the “Black & White” board game described on the Web site of the course. The following techniques were used during the design of this agent: the minimax algorithm with alpha-beta pruning, iterative deepening, and a transpositional table using Zobrist keys.

AI Programming Design

The minimax algorithm [2] that I implemented is actually and unknowingly a variant of what is sometimes called "negamax". In this form, the algorithm is reduced to a single recursive function instead of two and is useful only for zero-sum games, such as "Black & White", where if one player loses (utility = -1), the other wins (utility = +1), or if the game ends in a draw, the utility is 0 for both and the sum is again 0. My function, minimax(alpha, beta, depth) is implemented as a “max value” function, and implements alpha-beta pruning. It is called recursively in this manner: utility = -minimax(-beta, -alpha, depth+1), so that child nodes get negative values, and switched alpha and beta, effectively using the recursion as a “min value” function and back to a “max value” function for the next level of recursion.

However, since it is not possible to reach the leaf nodes of the game tree in any amount of tractable time, a call to an heuristic evaluation function at the cutoff depth is needed. These nodes are treated as leaf nodes after heuristic evaluation. My evaluation is made up of four features: the number of protected pieces (north-south and/or east-west, each protection counting as one), the number of doubly protected pieces (both north-south and east-west), the number of lined up pieces squared on each row, column and diagonal (squared to emphasize the fact that aligned pieces are more valuable than randomly placed ones), and the number of threats made on opponent pieces for possible capture on next turn. Weighted sums are computed for both the player and his opponent, and the sum of the opponent is subtracted from the one of the player, since a good value for the opponent is bad for the player. To find optimal weights for these features, I tried a simple training method where the computer is pitted against itself for days playing games with incrementally different sets of weights. My hope was to find an undeniably stronger set of weights (the global maximum), but alas, I would need to perform more comprehensive analyzes with reinforcement learning methods. Based on an intuitive analysis of the numbers and after a few trials, I settled on the following weights: 1, 1, 4 and 3 for the protection, double protection, line-up and attack features respectively. To demonstrate the performance improvement, making the computer play against itself, one side with the heuristic evaluation and the other without, the former always wins.

From these weights, it appears piece alignment is an important feature as hypothesized, but it is surprising to see that it seems threats are more important than protected positions. This underscores a possible fact that threats are an important strategy leading to "magical" alignment of pieces on the board, or maybe it is simply the fear of being threaten by the opponent that makes for a stronger survival feature.

Next, I implemented iterative deepening [2]. It serves more purposes than initially thought. First, it makes it easy to implement a time limited search. If the maximum allotted amount of time is used up, it simply aborts the current iteration and returns the results of the last completed iteration. Second, it helps the agent go to the shallowest goal state instead of possibly trying to reach a deep goal state if a shallower iteration was not performed previously. Third, each iteration is bootstrapped with the previously best evaluated successor node. Only the value of the first ply is used, but the transpositional table is not flushed. This helps the effectiveness of the alpha-beta pruning that is greatly enhanced if it has an initial good guess at the best successor node.

As an additional improvement and as my solution to the cycles problem, I implemented a transpositional table over a hash table using Zobrist keys [1]. Zobrist keys are calculated as follows. First, on startup or off-line, random numbers are generated for each positions on the board (25), and for each possible state (empty, white or black). It is also necessary to generate two other random numbers for the two players (one of which can be zero), for a total of 25*3+2 = 77 random numbers stored in an array. Then, for each node, the key can be computed as follows:

```
key = 0
for each of the 25 positions
    key = key XOR random_number[position][piece at position (empty, black or white)]
key = key XOR random_number[player of current ply (black or white)]
```

Therefore, these keys are quickly calculated (and even more quickly updated due to the reversible nature of XOR operations), and are sufficiently randomly distributed, if the initial numbers are sufficiently random. I use 64 bit keys in my implementation, but only the lower 23 bits are used for the hash code. The higher bits are stored as data in the hash table for collision detection. The rest of the data stored consists of: the "height" of the stored node (how deep of a tree does it represent), the evaluated value of the node, and a value type, indicating if the value is an exact value of the node, or if it is an upper or lower bound on the exact value. The latter cases happen as part of the alpha-beta pruning algorithm. Cycles produce the same nodes many times, but the alpha and beta values received from parent nodes usually differ, so by analyzing the value of the previous and current values of alpha and beta, it is possible to deduce new boundaries (values of alpha and beta) if needed [1]. This is a weak kind of “move ordering”, and helps the alpha-beta algorithm do its job on previously encountered nodes, even if it has not previously calculated exact values. Also, values for nodes with bigger “heights” than those stored in the hash table are not returned by the transpositional table. Doing so would result in reduced “intelligence”.

This transpositional table helps the program gain about one more ply. After 7 seconds on my old Athlon 1000MHz, the program frequently reaches the 9th ply without the transpositional table, but frequently reaches the 10th ply with it. It is not a huge improvement, and does not bring big improvements in game play either, but the effects are there and notable. To demonstrate the performance improvements, here are parts of the listings produced by gprof (an execution profiler) for a typical execution of the program with an equal cutoff depth of 8 for both executions with and without transpositional table:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ks/call  Ks/call  name
 45.43    470.32   470.32 188593084     0.00     0.00  Node::evaluateAs(Player)
 21.93    697.32   227.00 451501900     0.00     0.00  Node::movePiece(int, int, int, int)
 16.56    868.77   171.45      714     0.00     0.00  Node::minimax(int, int, int, Node*, bool)
 10.34    975.85   107.08 28920869     0.00     0.00  Node::getSuccessors(Node**)
  5.20   1029.64    53.79      102     0.00     0.01  Node::play(int, bool, bool)
```

Text 1 – Profiler results without transpositional table

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 36.65    203.69   203.69 79787900     0.00     0.00  Node::evaluateAs(Player)
 26.09    348.67   144.98      714     0.20     0.70  Node::minimax(int, int, int, Node*, bool)
 19.55    457.33   108.67 207789344     0.00     0.00  Node::movePiece(int, int, int, int)
  9.99    512.85    55.52      102     0.54     5.42  Node::play(int, bool, bool)
  7.19    552.82    39.97 12157093     0.00     0.00  Node::getSuccessors(Node**)
```

Text 2 – Profiler results with transpositional table

Clearly from those listings the work load by using the transpositional table has been cut by almost half (cumulative seconds), and a lot of the load has shifted away from the evaluation function evaluateAs(). This is to be expected, since a lot of leaf nodes do not need to be reevaluated frequently anymore. The effect is less pronounced on the other functions which still need to be called for non-leaf nodes: minimax(), getSuccessors(), movePiece() and play().

Concluding Comments

For this assignment, I programmed a “Black & White” game-playing agent based on the following artificial intelligence techniques: the minimax algorithm with alpha-beta pruning, iterative deepening, and a transpositional table using Zobrist keys. I designed as well an heuristic evaluation function making this agent along with the rest of the techniques a possibly strong player.

As shortcomings, the heuristic evaluation function is obviously not optimal, and can be optimized using better methods such as reinforcement learning. Better move ordering could also be implemented to optimize the alpha-beta pruning even further.

Although not directly related to artificial intelligence, the code is not optimal in terms of memory bandwidth. Nodes should be compacted in smaller representations that could be as well possibly computed faster. On the plus side, I used a dedicated stack of nodes to more easily manage my nodes using pointers, but with minimal overhead on creation and without relying on C++'s stack and its limited scope.

As a final note, it would be a good idea to add a maximum number of repeated moves in the rules of the game. When my program plays against itself, both sides frequently end up repeating the same move over and over again, one side usually sticking to the moves because it is the only way to prevent the other side from winning, while the winning side, of course, does not want to let go of such a high valued opportunity.

References

[1] Martin Fierz, *Strategy Game Programming - Advanced Techniques,* http://www.fierz.ch/strategy2.htm , 2002

[2] Stuart Russell and Peter Norvig, *Artifical Intelligence: A Modern Approach*, Prentice Hall, New Jersey, Second Edition, 2003.