CMPUT 657 Heuristic Search
Assignment 1 Report
Part 1

Talha Ibn Aziz
Student ID: 1669108

February 15, 2021

Essential Details

The program code has three essential files. They are namely "ataxx_state.cpp", "ataxx_state.h", and "main.cpp". The first and second files are related to each other and together form the "Class", which maintain the different states of the game "Ataxx". The third file contains the menu options and calls the different state functions, based on the options given (i, s, b, w, etc.). The search algorithms are also implemented in the "Class" files.

Board: The board is represented as a two-dimensional array of characters (w, b, e, x) as mentioned in the assignment question (a1.txt). All the searches are done within a state class without recreating the board again and again. The transition from one state to another is stored as a *move stack* containing only the change in the specific piece related to the move. Capture information is also stored to keep track of which pieces were captured and which pieces were unchanged.

Optimized Storage:

The moves and board are also represented as characters and unsigned long long integer (64 bits), respectively, for faster processing. Bit-wise operators are used to manipulate these variables.

For every location in the board (maximum $8 \times 8 = 64$) one bit is used to represent presence or absence of piece. As storing the obstacle positions is not necessary for Transposition Tables, two unsigned long long integers are enough for storing the white and black piece locations, respectively. A separate count for empty locations, white, and black pieces are also stored for faster calculation. All these variables are changed incrementally with each normal or undo move.

For each move, the necessary information is just the row and column numbers of the old and new piece positions. As the maximum possible value for the row or column is '8', 4 bits is enough to store it. Therefore, for 4 values (row and column of old and new location) 8+8 = 16 bits is enough. So the move stack has 3 characters for storing one move - first character (8 bits) and second character (8 bits) stores the move locations. The third character stores the capture information as follows: Each position adjacent to the new piece location is represented as an index of the *direction matrix* (explained later). There are 8 possible adjacent locations, so a character of 8 bits is enough to store this information. If the bit is 1, then that piece was captured, and the bit is 0 otherwise.

Move Generation: For generating valid moves and for capturing pieces, direction matrices are used. A direction matrix stores information for each possible adjacent location. For example the given Figure 1 shows the used direction matrices. The *row_d1* and *col_d1* arrays correspond to the adjacent positions in terms of row-number and column-number, respectively. An iterative loop can be used to traverse through the array to generate moves - both cloning and jump moves (Figure 1).

As cloning moves are usually better than jump moves, the jump moves come later in the array when looping through the generated moves. There are also checks in place for determining valid moves.

Experimental Section

The improvement of adding basic move ordering to perform clone moves first, reduces the search tree greatly. The difference can be seen specially in greater depths as shown in the given figures. The number of calls without move-ordering for a board state is 446, 1656, and 16411, in trees of depth 3, 4, and 5, respectively (Figure 2). On the contrary, for the same board state and search-tree heights, the number of calls are reduced to 375, 905, and 7562, respectively (Figure 3). The difference increases drastically with further increase in tree-depth.

```
/**
Direction matrices for piece capture
This is also used to store history
*/
int row_d1[8] = {+1,-1,0,0,+1,-1,+1,-1};
int col_d1[8] = {0,0,+1,-1,-1,+1,+1,-1};

///Direction matrices for both one and two-cell moves
int row_d2[24] = {+1,-1,0,0,+1,-1,+1,-1, 2,2,2,2,2, 1,0,-1, -2,-2,-2,-2,-2, -1,0,1};
int col_d2[24] = {0,0,+1,-1,-1,+1,+1,-1, 2,1,0,-1,-2, -2,-2,-2, -2,-1,0,1,2, 2,2,2};
```

Figure 1: Direction Matrices

```
Search Depth: 1000 Search Time Limit: 1
CTM DDD Time        Calls     Speed       TTQ       TTF       TTC  CBF    [P-Variation] Val
WHT   1  0.0           15       ---         1         0         0 ---- [a8b7 ---- ----]   1
WHT   2  0.0           62   20742.7        16         1         0  2.5 [a8b7 h8g7 ----]   0
WHT   3  0.0          446  111891.6        62        16         0  1.9 [a8b7 h8g7 a8b8]   1
WHT   4  0.0         1656  207622.9       441       138        78  3.4 [a8b7 h8g7 a8b8]   0
WHT   5  0.0        16411  864555.9      2266       764       388  4.9 [a8b7 h8g7 a8b8]   1
WHT   6  0.0        54645 1145717.6     13215      4635      3422  4.3 [a8b7 h8g7 a8b8]   0
WHT   7  0.7       2703136 3674092.0    323812     92630     81970  8.2 [a8b7 h8h7 a8b8]   1
TIMEOUT
WHT   8  1.0        977450  976484.3    461629    136090    117020  6.7 [a8b7 h8f8 b7a5]  -1
move: a8b7
```

Figure 2: Search Details without Move ordering

```
Search Depth: 1000 Search Time Limit: 1
CTM DDD Time        Calls     Speed       TTQ       TTF       TTC  CBF    [P-Variation] Val
WHT   1  0.0           15       ---         1         0         0 ---- [a8a7 ---- ----]   1
WHT   2  0.0           42   19186.8        16         1         0  1.0 [a8a7 h8h7 ----]   0
WHT   3  0.0          375   89584.3        58        16         0  1.0 [a8a7 h8h7 a8b8]   1
WHT   4  0.0          905  126044.6       433       186       130  1.0 [a8a7 h8h7 a8b8]   0
WHT   5  0.0         7562  534001.8      1377       704       414  1.7 [a8a7 h8h7 a8b8]   1
WHT   6  0.0        18572  615925.4      8307      4641      3856  1.5 [a8a7 h8h7 a8b8]   0
WHT   7  0.3       885131 3410107.8     96733     33835     29776 10.1 [a8a7 h8h7 a8b8]   1
TIMEOUT
WHT   8  1.0       2779314 2778344.4    570192    215598    202072  6.5 [a8a7 h8g7 a8b8]  -1
move: a8a7
```

Figure 3: Search Details with Move ordering

The decrease in function calls is primarily contributed to finding of Beta-cuts early. The number of moves considered in a state on average, before finding a Beta-cut is denoted by the CBF values. Therefore, observing the CBF values from both the figures - proof for the made statement can be found. The CBF value is higher without move-ordering and the value is lower with basic move ordering. However, for the tree depth of 7, the CBF value is higher for search using basic move ordering because, the best move for a state - seven moves into the search tree - is a jump move. Therefore, this move ordering helps prune the search tree significantly more when the best move is a cloning move (which is usually the case).

Further improvement of computation time is achieved through the use of state representations and comparisons using single variables (bitwise comparison) instead of comparing entire arrays. The move stack and transposition table is used efficiently in terms of both memory and time, using this technique. The best move of previous search from Transposition Table also reduces Search Tree further into the search. However, this has less effect for higher nodes that are closer to the root.

To reduce memory leaking, no dynamic memory allocation is used. Most variables are simple and efficient.

Implementation Correction:

While implementing the codes for Alpha-Beta Search with Transposition Tables, the limits set for table entries (EXACT, UPPERBOUND, LOWERBOUND) according to the Alpha-Beta return theorem, gave some errors. The checkTT function changes the bound (alpha, beta), whenever it finds the EXACT flag and sets the TT entry score as the bound. However, the Alpha-Beta call did not work properly in my program because - the Alpha-Beta search algorithm could not find a better move compared to the TT entry score and the move according to the previous search is returned (wrong answer). Therefore, I changed the alpha-beta range of the pseudo-code in the slides to $alpha \leftarrow TTscore[idx]$ and $beta \leftarrow TTscore[idx] + 1$ instead of $alpha \leftarrow beta \leftarrow TTscore[idx]$ so that even an equally better move is selected, when found.

Implementation Variations

There are a few implementation specific handling of cases:

- If the game is already over and a manual or automatic move is given, the program takes the input and processes it. After processing the command, only then it prints that the game is over, shows the score and refuses to search or accept any board-specific commands until the board is setup.

List of Sources

1. **Slides Provided by Professor Michael Buro: Part 2 and Part 3:** I learned the main algorithms (MiniMax, Alpha-Beta, using Transposition Tables, etc.) from the lectures and slides.

2. **Website:** en.cppreference.com, I used this website to refer to known and unknown functions and libraries when required (while I was writing the code).

3. **Website:** https://www.geeksforgeeks.org/, I learned the Zobrist Hashing technique from this website.