CMPUT 657 Heuristic Search
Assignment 1 Report
Part 1

Talha Ibn Aziz
Student ID: 1669108

February 8, 2021

Essential Details

The program code has three essential files. They are namely "main.cpp", "ataxx_state.h", and "ataxx_state.h". The first file contains the menu options and calls the different state functions, based on the option given (i, s, b, w, etc.). The second and third files are related to each other and together form the "Class", which maintain the different states of the game "Ataxx". 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 move 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) for faster processing. Bit-wise operators are used to manipulate these variables.

For every location in the board (maximum of $8 \times 8 = 64$) 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 for faster calculation. All these variables are changed incrementally with each normal and undo moves.

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* array corresponds to row-wise and the *col_d1* array corresponds to the column-wise adjacent positions. A loop can then be used to traverse through the array to generate moves - both cloning and jump moves (*row_d2* and *row_d2*).

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.

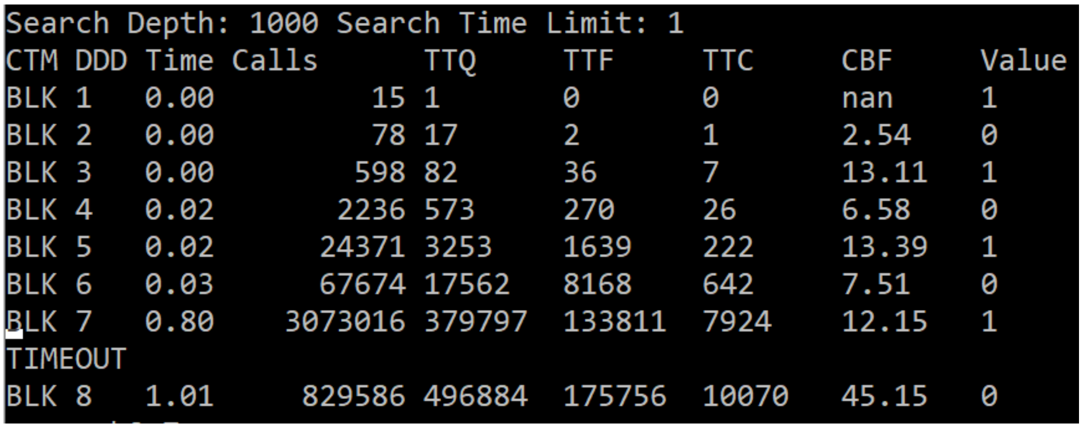
Experimental Section

The improvement of adding basic move ordering of performing of 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 598, 2236, and 24371, in trees of depth 3, 4, and 5, respectively (Figure 2). On the contrary, for the same board state and heights, the number of calls are reduced to 461, 1279, and 9290, respectively (Figure 3). The difference increases drastically with increasing tree-depth. The difference in calls also decreases time consumed for the Alpha-Beta search as represented

```
/**
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      TTQ     TTF     TTC     CBF     Value
BLK 1   0.00        15  1       0       0       nan     1
BLK 2   0.00        78  17      2       1       2.54    0
BLK 3   0.00       598  82      36      7       13.11   1
BLK 4   0.02      2236  573     270     26      6.58    0
BLK 5   0.02     24371  3253    1639    222     13.39   1
BLK 6   0.03     67674  17562   8168    642     7.51    0
BLK 7   0.80   3073016  379797  133811  7924    12.15   1
TIMEOUT
BLK 8   1.01    829586  496884  175756  10070   45.15   0
```

Figure 2: Search Details without Move ordering

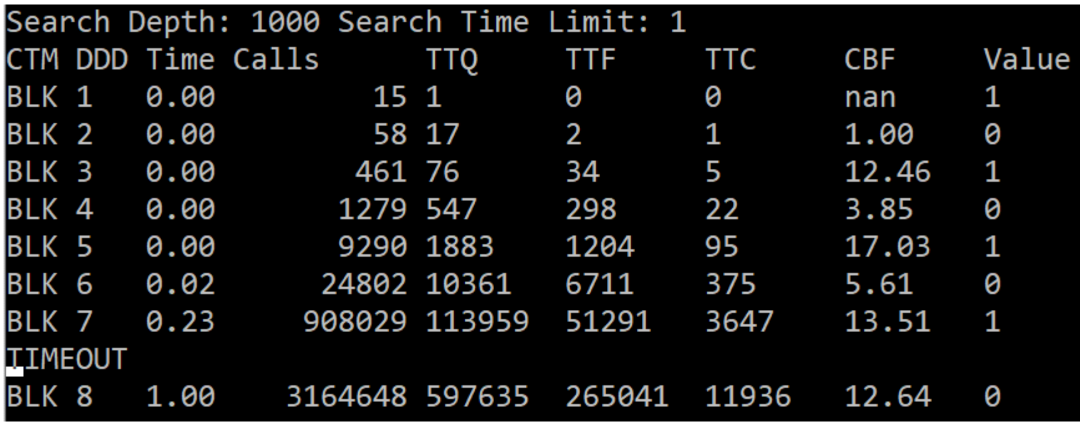
in the mentioned figures.

Further improvement of computation time is achieved through 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 which 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 which changes the bound (alpha, beta), whenever it found the EXACT flag and set the TT entry score as the bound, the Alpha-Beta call did not work properly. It was because - the Alpha-Beta 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 $alpha \leftarrow TTscore[idx] - 1$ and $beta \leftarrow TTscore[idx] + 1$ instead of $alpha \leftarrow beta \leftarrow TTscore[idx] - 1$ so that even an equally better move is selected, when found by



```
Search Depth: 1000 Search Time Limit: 1
CTM DDD Time Calls       TTQ     TTF     TTC     CBF     Value
BLK 1   0.00         15 1       0       0       nan     1
BLK 2   0.00         58 17      2       1       1.00    0
BLK 3   0.00        461 76      34      5       12.46   1
BLK 4   0.00       1279 547     298     22      3.85    0
BLK 5   0.00       9290 1883    1204    95      17.03   1
BLK 6   0.02      24802 10361   6711    375     5.61    0
BLK 7   0.23     908029 113959  51291   3647    13.51   1
TIMEOUT
BLK 8   1.00    3164648 597635  265041  11936   12.64   0
```

Figure 3: Search Details with Move ordering

Alpha-Beta search.

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.