yiannig-rdelaus-cburnett

project 4 write up

In order to assess the least efficient areas in the code base, we first thoroughly addressed lecture slides and all README’s to gain an understanding of the general structure. We then read through the code base and determined that we would focus on maintaining heuristics and just improving speed. We decided to potentially return to heuristics in the future to make some sort of compromise between accuracy and speed, but didn’t know enough about the game of Leiserchess to make any assumptions about which heuristics could be omitted or stripped down in some way.

After using perf to retrieve profiling data on the reference implementation, we noticed two areas within the code base that suffered from major inefficiencies. The first area is the representation of the board. The second area is the algorithm used to determine the laser\_coverage heuristic. Given that we have three people, we agreed that we would be most effective if two people pair programmed on one task and the third person worked individually on a separate task. We predicted that changing the representation of the board would be a more difficult task that modifying the laser\_coverage algorithm, and decided that Rob and Cameron would spend their time pair programming the board rep modification task while Yianni worked to optimize the laser\_coverage calculation implementation.

We noticed that in eval and low\_level\_make\_move, loops are made through the entire board to find and make changes to pieces. We postulated that modifying the representation of the board from square-centric to piece-centric would allow for the client to find pieces by simply looping through pieces. Since there are significantly more locations on the board than pieces, implementing this new board representation would greatly reduce loop overhead when finding pieces to modify.

To implement this change from a square-centric to piece-centric board representation, we originally decided to augment the position struct with a 2d array of integers which mapped to indices in another array of location\_t objects. Each location\_t stores a row, column, and boolean “isPiece”. As we read in the initial board position, when a piece was encountered, we added the piece’s location to the array of location\_t objects and then stored the index into this list in the 2d array in position. The 2d array had size BOARD\_WIDTH x BOARD\_WIDTH and mirrored the board, with empty squares having value -1 and squares with pieces having the value of their location in the piece locations list. Then, rather than looping over the entire board to find pieces, we simply looped through the list of piece locations. To update these data types as pieces are moved, we used our 2d array to find the index of the piece in our list of locations and then updated this location value as well as the index-storing 2d array. As pieces are deleted, we simply changed their values in the 2d array to -1, indicating they no longer lie on the board. While we augmented the position data type with our new board representation, we maintained the original board of pieces as well, since many of the original functions relied on this representation.

We thought this new implementation would provide a significant speedup over the reference. Unfortunately, this was not the case. After several hours of debugging, we found our new version was actually slower than the reference. We continued looking for issues in our code, thinking we hadn’t taken full advantage of the potential speedup this improvement could make. However, after profiling our code, we noticed that the mem\_copy\_avx\_unaligned was incurring significantly more overhead than in the reference implementation. We think this inefficiency was caused by the extra overhead required to update the 2d index board at every position update. After much consideration, we decided to ditch this original idea and think of a new way to establish a piece-centric board representation.

Our new idea was to get rid of the 2d index array and store the index of pieces in the uppermost bits of the piece\_t datatype. This eventually provided the optimization we were originally expecting, but unfortunately we were unable to finish debugging before the beta I deadline. This new implementation will be discussed further at the end of this report.

After thoroughly analyzing the algorithm used to determine laser\_coverage, we realized that the algorithm traverses over every possible move, given a board state, to determine the coverage score. Although this will always work, if a move does not affect the original path of a laser, the coverage score will not be affected. Given this fact, we limited the computation needed to produce the laser\_coverage score by only recomputing the laser\_coverage score if a move modified the original path of the laser. We did this by using the mark\_laser\_path function to produce the path of the laser before and moves have been calculated. We then calculated a move and check to see if that move intersects the original path in O(1) time. Although we needed to add an if statement into the body of the loop that traverses over all the possible moves, this cost of this extra branch is significantly outweighed by the gains of performing unnecessary computations on moves that do not affect the path of the laser.

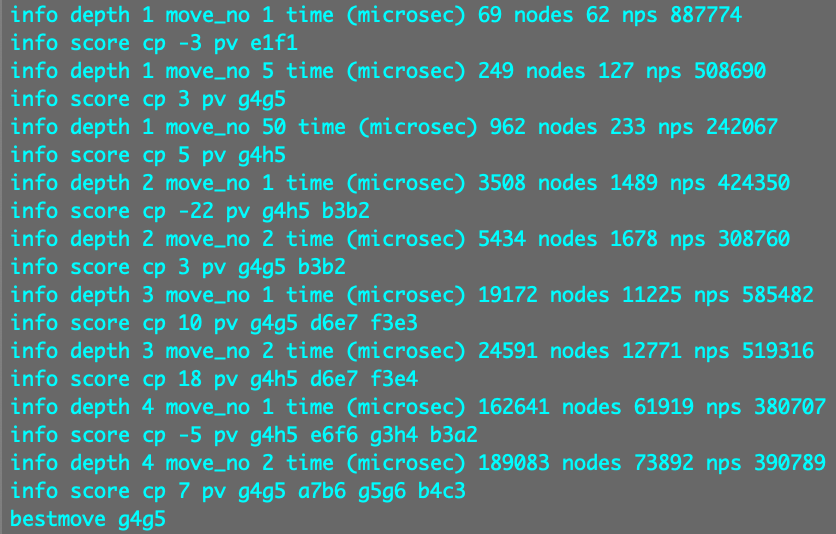
Due to debugging time, we were unable to submit a version of our solution before that deadline that optimized the board representation. Our most recent commit in master was pushed after the Beta I deadline and contains a new board rep that does significantly improve speed. This board representation utilizes an array in positions where each index is designated to a piece and a square\_t is stored at each index, representing the location of that piece. We pack a uint7 representing the piece’s index in the piece\_t type. When a piece is moved, we used the index stored in its type to access the index of the locations list in position in O(1) time. The location at this index is then updated according to a change in the pieces position. This implementation used significantly less overhead than our original implantation because this implementation no longer required the maintenance of an addition 2-d array.

Another modification we made to improve efficiency was changing the ARR\_WIDTH size from 16 to 10. Since we were not utilizing the additional space outside the board for sentinels, by decreasing the size of ARR\_WIDTH we decreased the number of spaces need to search through the board and reduced the overhead of several loops throughout the code base.

Unfortunately, we did not accomplish as many optimizations as we wished for Beta I. For Beta II, we plan to make additional serial optimizations before attempting to parallelize our code. We noticed that the current implementation is not very suitable for parallelism because our board rep is not thread safe.

**Profiling Data:**

Original implementation:

A screenshot of a computer

Description automatically generated

After new board rep:

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated

After implementing optimized laser coverage heuristic (final Beta I implementation):

A screenshot of a cell phone

Description automatically generated

A screenshot of a computer

Description automatically generated