6.172 Project 4 Final Write-up
agrebe, crinard, jaytlang, adriano

Table of Contents:

Executive Summary

We were given an engine for playing Leiserchess, a variant of laser chess. The original implementation contained many algorithmic strengths, including alpha-beta pruning and caching board states, but many of the low-level routines, especially for evaluating heuristics of board state, were unoptimized. We were tasked with optimizing the engine enough to be competitive with other staff implementations, including one designed to run five times faster.

Our high-level algorithmic design is fundamentally the same as the given implementation. Our main change for the beta was to increase the speed of the laser-coverage heuristic called during board evaluation. We also implemented an opening book that pre-computed common opening sequences and the best responses to them. For the final we re-tuned the heuristics' weights and dropped some expensive heuristics to allow for deeper searching.

We also attempted various changes which were less successful. These include a bit-board, a "branching" laser coverage function, various attempts to parallelize effectively, bitmaps of different kinds, Bentley-rule optimizations (probably already caught by the compiler), different size boards, and more. We also had some ideas we did not implement, including a graph data structure for faster laser traversal.

Bottlenecks Over Time

Originally we found that laser coverage was one of our biggest bottlenecks, as well as *__memmove_avx_unaligned_erms*, due to copying of the board which was unnecessary. We improved our runtime significantly by passing pointers to the board and editing the board instead of copying it and optimizing laser coverage as much as possible, as well as other Bentley optimizations in other parts of the code. Another thing that significantly boosted ELO was including an opening book.

From perf, about half the runtime in the beta submission was in computing laser_coverage and the add_laser_path subroutine that it calls. About 15% of runtime is spent in move generation for the laser coverage, about 10% of runtime spent in scout_search (a significant fraction of which is move sorting), and the other 25% of runtime is in miscellaneous function calls. Optimizing add_laser_path was a significant goal for the final, but given that we have already studied this code, we reasoned that there was unlikely to be low-hanging fruit remaining and further optimizations would require more effort. A diagram of bottlenecks as of the beta submission (identified by pprof) is pictured in Figure 1.

For the final, we tried a number of different approaches to parallelizing, but found that we had very high parallelization overhead and very high node counts. Because of these factors, we found that the effects of parallelization were actually worse (even on 64 cores). We managed to generate a better opening book, significantly alter and improve the laser coverage heuristic, and optimize out a fair amount of other work.

Beta Performance Optimizations

A significant bottleneck in the original code which took roughly a quarter of the runtime was copying the board at each move, and in particular for each ply during laser coverage computation. This could be avoided by making many of the moves in place (without copying the board) and then undoing them afterwards, saving about 25% of runtime. We also realized that about half the moves considered during laser coverage had no effect on the laser path, so adding the laser path on these moves could be avoided, saving close to half of total runtime. Enabling link-time optimization gained us about 10% of runtime. Overall, this gained about a factor of four in performance relative to the original implementation.

Opening book

From a limited sample of watching games in the GUI, it appeared that the bots spend about a quarter of their total allocated time in the first half-dozen moves by each side, and in this time, they only managed to reach depth 5 or 6 and therefore made sub-optimal moves. (In particular, the best opening move at depth 9 is b4c3, but this is not found at the depth 5-6 searches typically run.) Since the initial position is fixed, common opening sequences can be hard-coded into the bot in an opening book. Our beta submission had an opening book of common openings out to about 10 ply, with the initial moves computed to depth 8 and subsequent ones computed to depth 6. For most opening lines, this makes our bot both better and faster. Empirical playtesting found that the version with an opening book outperformed the one without by about 150-200 Elo, which was a significant factor in our ability to beat the reference bot.

Final Performance Optimizations

Opening Book Improvements

For the beta submission, the opening book was nominally computed out to a depth of 10 ply. However, some common opening sequences were missing, and others were only

evaluated to a low depth. For the final submission, we computed the book out to 13 ply and ensured that all of our moves were computed out to at least depth 8. We also automated scripts to ensure that the book was complete (since our generation scripts were prone to various failures and the outputs needed to be verified post hoc). The deeper and more accurate book increased Elo by about 50.

Eval Optimizations

We saw that a major part of the eval() function was in calculating the NMAT and PMID heuristics. We were doing a double for() loop over every square of the board to find pawn locations, where if we kept a pawn-centric data structure as well, we could avoid this. We added an array of the two sides' pawn locations to help calculate PMID, and we rolled the NMAT calculation into the laser coverage function, saving us another 2 computations of the laser path per eval() call. These optimizations improved nodes/sec by about 10%, and added about 15 Elo compared to our previous version.

Eval Heuristic Retuning

Inspection of other team submissions showed that other teams had had success with changing the default weights in the static evaluator. Ideally, we would retune weights via playtesting of different combinations, but playtesting requires very high statistics in order to see a significant change, so it is computationally expensive to try more than a handful of parameter choices.

Instead, we took a sample of about 1000 games that we had already run and looked at every position in each of the games. We then computed each of the heuristics in the static evaluator at the various positions to see how well the heuristics could predict whether the position was a win (+1), loss (-1) or draw (0). One way to determine the predictive power of the various heuristics is a multivariate linear regression of the output of the game as a function of the various heuristics, and the coefficients of the regression give a set of weights for the heuristics. The weights from this regression differed markedly from the defaults in the static evaluator. In fact, a couple coefficients (those of PPROX and PTOUCH) were negative, and we decided through playtesting to set these weights to zero in our final fit. The reweighted evaluator (now with two fewer heuristics) beat our beta submission by about 200 Elo.

We then decided to push this idea even further. The laser coverage computation (and its subroutines) was a very expensive computation, taking about two-thirds of our beta submission's runtime. We tried to fit our data by simply removing the laser coverage heuristic entirely (and retuning other weights to compensate), but the threefold gain in speed was offset by the decreased predictive power of the static evaluator with laser coverage removed. A compromise solution was to use an approximation to the original laser coverage heuristic that measured only the current laser path (and not all paths reachable within one ply). This gave us most of the performance benefits of removing laser coverage while maintaining decent predictive power from the static evaluator, giving us an additional 50 Elo in performance.

Calculating Laser Paths Less

We realized that the laser path as calculated by the gravity laser was identical to that calculated by the laser coverage functionality. We decided to merge them to avoid having to calculate the same directions multiple times. This gave us a small boost of around 10 ELO, more or less.

Packing Data Structures

For the beta submission, our computational cost was dominated by calls to laser coverage, so our data structures had to be optimized for fast accesses during computation of the laser path. We had tried packing the structs into fewer bytes, but this slowed laser path computation and was originally a net negative. Once we had removed most of the laser path computation, access times during laser path computation were no longer as important, and copying times became relatively more of a bottleneck. As such, adding compiler directives to pack the structs into less space became a net speedup of about 5% in the final submission.

Parallelization Results

After our beta submission, we attempted to parallelize alpha/beta search as well as the zero-window scout search following from various algorithms and tips provided during lecture. Our first efforts involved an implementation of Young Siblings Wait (YSW); surprisingly, when applied to both principal variation and scout search, this did *not* produce a statistically significant advantage over serial code under most tests.

One reason parallelism may not have been effective is high variability in search node scheduling order within Cilk. For example, in naive YSW on an almost-best-ordered game tree, if the second child in traversal order is the best choice of child, it will be searched in parallel with other non-first children. With our understanding of the Cilk work-stealing scheduler, we conjectured that there was no surefire mechanism to evaluate that second child before later children, as the order of cilk_spawn invocations rarely reflects scheduler behavior. As such, runtime is highly dependent on game tree ordering as well as non-deterministic scheduler operation during a particular game; the latter is largely out of our control.

Our hypothesis is corroborated by extremely high variability in runtime we observed under the parallel searches - simple runs to a given depth (e.g. `go depth 8`) varied by a factor of up to four in our tests. Because of this, much of our effort was spent on engineering ways to reduce scheduler variability - trading some parallelism to get a more consistent runtime. We developed three *tuning parameters* by which we tried to fulfill this goal - *n*-siblings wait, a coarsening depth *d*, and a batch size abstraction *b*.

N-Siblings Wait

For almost-best-ordered game trees where the nth (where $n << tree\_breadth$) child is often the best choice, we can use *n-siblings-wait* to capture this state efficiently. Our

algorithm mirrors those seen on recent quizzes - *n* children are executed and evaluated serially, and the remaining children are executed in parallel as in typical YSW. The choice of *n* becomes an important tuning factor - higher choices of *n* trade parallelism for lower variability.

Coarsening Depth

One trivial optimization we can make to a parallel search is to *coarsen* recursion at sufficient search depth – we define a coarsening depth *d* at which point we execute all nodes serially. We find that setting reasonable values of coarsening depth universally benefits parallel runtime, but doesn't resolve issues of variability higher up the game tree. However, if the coarsening depth is chosen too high up the tree, we lose parallelism, and if coarsening depth is too low thread scheduling overhead dominates our runtime.

Batch Size

During the parallel execution phase of n-siblings-wait, one mechanism to eliminate variability is to schedule a fixed number of nodes to execute at a time. We define a batch of nodes to have size *b*, and execute batches of nodes in parallel in traversal order. This severely restrains parallelism, but we believed this had the potential to curb variability drastically, since we can ensure that execution order adheres more closely to traversal order. In our experiments, however, often the losses from extreme tail latency at the end of batches coupled with reduced parallelism made small batch sizes perform poorly.

Taken together, all of these parameters presented us with an opportunity to perform tuning. We wrote a script (player/search_params.pl) which allowed us to test runtime of searches to a fixed depth, and ran all reasonable combinations of parameters overnight in conjunction with serial code. All competitive parallel parameter sets were then played against each other in real-time Leiserchess matches. Disappointingly, even the best parallel parameter sets failed to yield a statistical advantage over serial code, owing to high variability, and high scheduling overhead even after reductions in variability. Thus, we elected to drop parallelism for our final submission, instead opting to spend time on the aforementioned static evaluator optimizations on a serial game bot.

Other Failed Optimizations

Beta

10x10 Board and Bitboard

We spent a lot of time and effort on trying to find faster board representations. The first thing that we tried was using 10x10 as opposed to 8x8. The 10x10 version We built a shim layer to abstract away the board representation from its behavior and replaced calls in the entirety of the codebase to use this shim layer. This made it easy to try different board sizes as well as a bit-board representation of the board using three

uint128s. This bitboard might work well with one of the optimizations that we tried (and sadly deleted the branch for) that iterated through the number of pawns for the laser path, after calculating a bitmask for where mirrors exist. When written without the bitboard, this algorithm took 1.2x the time of the original code to run. The distribution of the runtime of this was evenly distributed across instructions, indicating that the algorithm didn't have any easily identifiable bottlenecks. Of the runtime in the new add_laser_path, around 38% came from computing the bitmap, which could be gotten rid of using some of the architecture from the bitboard branch, but in the end it would only speed up by 10-15%. This may be worth revisiting if we decide that the parallelization of eval is something worth working on. On the bitboard, the time with using a drop-in replacement was approximately 1.3x the default board representation. Later on we tried another bitmap (inspired by rank zero) using 64-bit words, but we did not push it to completion, since we prioritized parallelization for the final, as well as lower hanging fruit in the serial side of things.

Graph Board Representation

We also searched for a graph representation of the board that would allow for easier laser coverage, in the end we were unable to come up with a conceptualization that was effective for the operations that we needed. Our best idea was to store, for each pawn and monarch, the closest pawn or monarch (or wall) in each of the four directions. This would make it easier for us to do certain aspects of laser coverage since we would not need to iterate through empty cells. However, it complicated other aspects, such as updating the graph, calculating the shortest distance to each non-pawn, non-monarch square hit by the laser, etcetera.

Short Circuiting Eval

We also tried to short-circuit eval to avoid calling laser coverage. The idea was that if the other heuristics were highly predictive of laser coverage in certain cases, there was no need to calculate it. For example, if the other heuristics gave the board a very, very low value, it is unlikely that laser coverage was necessary, and the same is true for a very, very high value. We tried to do a correlation analysis to see whether this would be successful. We had bugs and did not succeed. However, it did not end up mattering since we abandoned laser coverage for the final in favor of a simpler, single-node alternative. Instead, as we've outlined above, we pursued a strategy of tuning the weights for the different heuristics (even dropping some).

Branching within Laser Path Calculation

Finally, the current implementation first generates a list of moves and then calls add_laser_path on each of them. One idea was to walk along the laser path and execute moves as they are relevant to the path of the laser (i.e. moving pieces in or out of the path), since this would reuse the first half of the path rather than recomputing the entire path for each move. However, the attempted implementation of this includes the original path in the set of paths considered in the laser coverage (rather than just modifications to the original path), a small change but one that meant that existing

correctness tests were insufficient to evaluate this change. Playtesting against the previous implementation was inconclusive, so this change was not merged into the main branch.

Final

Bitmaps

We tried using bitmaps for the locations of pawns and of the laser path. This was meant to help us iterate only through pawns in eval, thereby reducing the work by skipping empty squares. We used constant time operations to get the leader zeroes and thereby find the location of pawns in the bitmap. This did not help so much as hinder our runtime, since, we think, it was deleterious to vectorization (by using a while loop instead of for loops). However, we never confirmed this. We tested primarily using `go depth 8` and consistently had worse times by around 1.3x.

Branchless Laser Traversal

There is a switch statement in laser coverage which checks the type of each cell it traverses to decide what to do. We had an idea to remove that by storing “accelerations” which you can add to your vertical and horizontal “velocity,” thereby making the code branchless. However, we never tried it since it was complicated and the switch did not appear to be a significant bottleneck.

Integer Arithmetic for Laser Coverage

Knowing that floating point operations could slow down performance, we changed laser coverage to do integer arithmetic. However, we ended up abandoning laser coverage in its entirety and so had close to no need for this optimization later on, which is why it is not present in our final deliverable. Moreover, we did not find it to be statistically significant as an improvement.

Tables of Tables of Tables

We tried creating a table for every conceivable calculation in eval. We used tables to do all floating point arithmetic possible. It gave us a statistically insignificant boost, and we were worried that as we added more and more it may actually start to matter how much memory we were taking up. We ended up ditching this change in part for those reasons, as well as the fact that it made the code less legible, and that we abandoned laser coverage, meaning we no longer had such a dire need for the table.

Tuning the Transposition Table

Early on we tried tuning the transposition table associativity size *k* and we found no meaningful connection between its size and performance (below a threshold). Above a threshold, (16) we found that it decreased performance.

Different Sorting

We tried to improve sorting performance by using merge sort, as well as by doing a partial selection sort (i.e. filling the first *k* elements of the array with the maximums). We found that neither was really effective. We tried merge sort at the 11th hour and had no time to fully debug its correctness, and it did not deliver real results. We think this was because the arrays it sorts are not that small, making the $O(n^2)$ factor for insertion sort not that meaningful to optimize against. We are not sure why partial selection sort was not that good. Probably, the keys were not as indicative of board quality as we would have liked.

Team Dynamics

For the beta, Chris and Adriano did a lot of peer programming. Jay focused on tests and was getting ready to work on the main project when he had to leave for a family emergency. Chris and Anthony met frequently to discuss next steps and bounce ideas off of each other. Anthony did most of the work on making eval fast asynchronously, with some peer programming with Chris. As a team, we met frequently to discuss improvements made and steps moving forward. Most people knew the areas that needed to be improved and were kept updated on areas of potential improvement.

For the final Adriano kept focusing on laser coverage and eval. The rest of the team initially focused on parallelization of search, but after a little while Anthony made great strides to improve eval by using a linear regression on past games to retune the weights. On the last couple days, after failing to get significant speedup for parallelization, the entire team went back to optimize eval and made various Bentley optimizations before submitting.

Individual Contributions

Anthony: For the beta, searched for bottlenecks with perf and found and eliminated several significant bottlenecks in laser coverage. Also generated a simple opening book and created scripts that can be used to extend the opening book. Tried unsuccessfully to perform branching within laser path computation to eliminate the call to move generation. Ran and analyzed play-testing of various ideas to determine which ideas were net improvements. For the final, recomputed the opening book and fixed broken lines. Also found a better set of parameter weights for the static evaluator and created a drastically simplified approximation to laser coverage that worked with retuned weights. Worked unsuccessfully on parallelism, both individually and with group members.

Chris: For the beta, pair-programmed with Anthony early on in the project to get easy gains (editing rather than copying board). Worked on for() loop implementation of laser coverage, while(1) {head->wall} on clever laser coverage algorithm, first pass on

short-circuiting, bitboard and different board implementations. For the final, did the initial exploration of parallelization, as well as trying out a substantial number of other approaches to attempt to get the “low hanging fruit”, then focused on serial improvements. These include trying to do a different bitboard representation, trying to do a version of laser coverage that cached the laser path, and eliminating heuristics, and making eval() substantially faster, resulting in an overall 20% speedup, and +24 -0 ELO gain over our previous bot.

Adriano: For the beta, pair programmed with Chris to try and implement bit-board. Tried to make changes to laser coverage to use integers instead of floats. Wrote documentation and lots of pseudo-code for different algorithms. For the final, tried various different flavors of bitmaps, sorting optimizations, bentley optimizations by using tables for different floating point values used by the heuristics, and branchless code (although this was unsuccessful).

Jay: For the beta, design document, unit testing infrastructure, consistency testing infrastructure. Wrote considerable documentation for various different dev-ops tasks integral to the rest of the team’s capabilities. Opened up a 64-core machine for testing during the final project phase, started transition into programming for the parallel project.

Project Logs

Jay:

- 11/10: team contract, 1 hr
- 11/13: meet with folks to do overview, play leiserchess, wrote unit tester - 6 hr
- 11/15: implement CHECK_REP and DUMP_DS subsystems, write design doc - 7.5 hr
- 11/20: implement consistency checking infrastructure + script ASAN/MSAN - 6 hr
- 11/21: salvaged + opened up 64-core machine for testing use by other group members - 3 hr
- 11/29: beta write-up - 2.25 hrs
- 12/03: help with initial attempts at parallelism - 6 hr
- 12/04: assist with laser coverage reduction, further experiments with retuning parallelism across various configuration parameters, pair programming - 6 hr
- 12/05: continuation of the above remotely - 2 hr
- 12/06: group attempts to parallelize code, presentation writing - 8 hr
- 12/07: presentation practice, gave presentation - 3 hr
- 12/08: read into static evaluator work + finalize parallelism attempts - 2 hr
- 12/09: final write-up - 1 hr

Adriano:

- 11/10: Team contract (1 hr)
- 11/13: Meet with team to do overview, play leiserchess (1.75 hrs)
- 11/14 & 11/15: Unclear; did not log what I did, maybe something hopefully not nothing (4.25 hours)
- 11/18: Reading the code, System diagram on paper, documentation (2.25 hrs)
- 11/19: Documentation centralization, OH, float to int for laser coverage, designing data structure with Chris (7.5 hrs)
- 11/20: Pseudocode for bit-boards and branchless laser traversal, call with Chris, created a log in Github wiki (4 hrs)
- 11/21: Reviewed Shim, planned w/ Anthony (1.25 hrs)
- 11/23: Working on 8x8 board, modified shim but was unsuccessful (6 hrs)
- 11/24: Working on short circuiting correlation and bitboard with Chris (7 hrs)
- 11/29 Beta Writeup (2.25 hrs)
- 12/03 Read through rank zero's code (0.5 hr)
- 12/04 Bitmaps, Bitboard (4.5 hrs)
- 12/05 Bitmaps, Path Optimization (2 hrs)
- 12/06 Laser Coverage, Bitmaps (5.5 hrs)
- 12/07 Heuristics (1 hr)
- 12/08 Optimizations to Eval, Trying to figure out parallelization, tests, collaborative work (9 hr)
- 12/09 Final Writeup (0.5 hr)

Chris:

- 11/9: Check out codebase (0.75 hr)
- 11/10: Team contract, pair working w/ Anthony (2 hr)
- 11/13: Meet with team to do overview, play leiserchess, start messing around with board representations (4 hr)
- 11/15: Meet with team to go over design submission, do final edits and formatting, submit (4.5 hr)
- 11/17: Pack board into 10x10 (failed 8x8 attempt), troubleshoot correctness and performance measurement issues (6 hr)
- 11/18: Rewrite 10x10 board rep, try to figure out how to validate (5 hr)
- 11/20: Figure out how to validate 10x10 rep, discover it has no performance impact, discuss branchless laser traversal and boards to support this (5 hr)
- 11/21: 11/20: Head->wall on closing book concepts (3 hr), build shim layer for boards (6 hr)
- 11/22: MITPOSSE meeting (1 hr), finish shim layer, start server testing and comparing branches with correctness-breaking optimizations (3 hr)
- 11/23: Work with Adriano on short-circuiting and bitboard (6 hr)
- 11/24: Individual work on bitboard, then work with Adriano & Anthony on bitboard, then short-circuiting, discuss final ideas (8 hr)

- 11/30: Initial parallelism (toss a cilk_for in scout search and send it) (3 hr)
- 12/1: YSW with scout search and testing (4hr)
- 12/4: Look at other beta code for ideas, quick prototyping (4hr)
- 12/5: Same as 12/4, (3hr)
- 12/6: Parallelism, caching lasercov (12 hr)
- 12/7: Parallelism, caching lasercov, presentation make, practice, set up AWS instance to run more tests (12hr)
- 12/8: Parallelism, AWS testing, Eval() optimizations w/ Adriano and Anthony (11hr)
- 12/9: Wrap-up, writeup (2hr)

Anthony:

- 11/8: Becoming oriented with project and codebase (1.5 hr)
- 11/9: Continuing to become oriented with codebase (1.5 hr)
- 11/10: Initial group meeting, preliminary search for low-hanging fruit for optimizations (4 hr)
- 11/11: Generation of rudimentary testing suite, optimizing codebase (4.5 hr)
- 11/13: Group meeting, individual programming (4.5 hr)
- 11/14: Worked on initial optimizations and merged these into master branch (6.5 hr)
- 11/15: Worked on design document, some programming (3.5 hr)
- 11/17: Individual programming (3.5 hr)
- 11/19: Started working on opening book (3 hr)
- 11/20: More work on opening book, attempt to find more performance optimizations (5 hr)
- 11/21: More work on opening book, attempt to find more performance optimizations (6.5 hr)
- 11/22: More work on opening book, playtesting of various branches (3.5 hr)
- 11/23: Analyzing results of playtesting, continuing to work on opening book, attempt to branch within add_laser_path (4.5 hr)
- 11/24: Minor optimizations, meetings to finalize beta submission (6 hr)
- 11/25: Opening book generation (1 hr)
- 11/26: Opening book generation (1 hr)
- 11/27: Opening book generation, other serial optimizations (2.5 hr)
- 11/28: Opening book generation, other serial optimizations (4.5 hr)
- 11/29: Opening book generation, other serial optimizations (2 hr)
- 11/30: Opening book generation, other serial optimizations (1.5 hr)
- 12/2: Opening book generation, other serial optimizations, initial attempt at parallelism (2 hr)
- 12/3: Meeting and individual programming to try parallelism (5.5 hr)

- 12/4: Computing better weights for static evaluator, trying versions with laser coverage heuristic eliminated or reduced, performing additional optimizations enabled by the reduction in laser coverage time (9 hr)
- 12/5: Additional work on retuning heuristics and determining which version of tuning weights to use, more attempts at parallelism (9 hr)
- 12/6: Group and individual attempts to parallelize code, writing presentation (5.5 hr)
- 12/7: Rehearsing and giving presentation, more parallelism attempts (8 hr)
- 12/8: Final attempt at parallelism, peer programming of simplifications to static evaluator (12.5 hr)
- 12/9: Final writeup (1 hr)

Total time: 123 hr

Acknowledgements

We would like to thank the course TAs for helping us come up with ideas for the beta. Specifically, Sameer, and Obada helped Adriano explore different theoretical options for speeding up laser coverage by changing the board rep. The lectures by course staff were also extremely helpful both in understanding the engine algorithms and in developing ideas for parallelism. We also want to thank rank zero for their idea to retune the weights in the static evaluator, as well as their ideas to use bitfields (even though those were less successful).

Figures

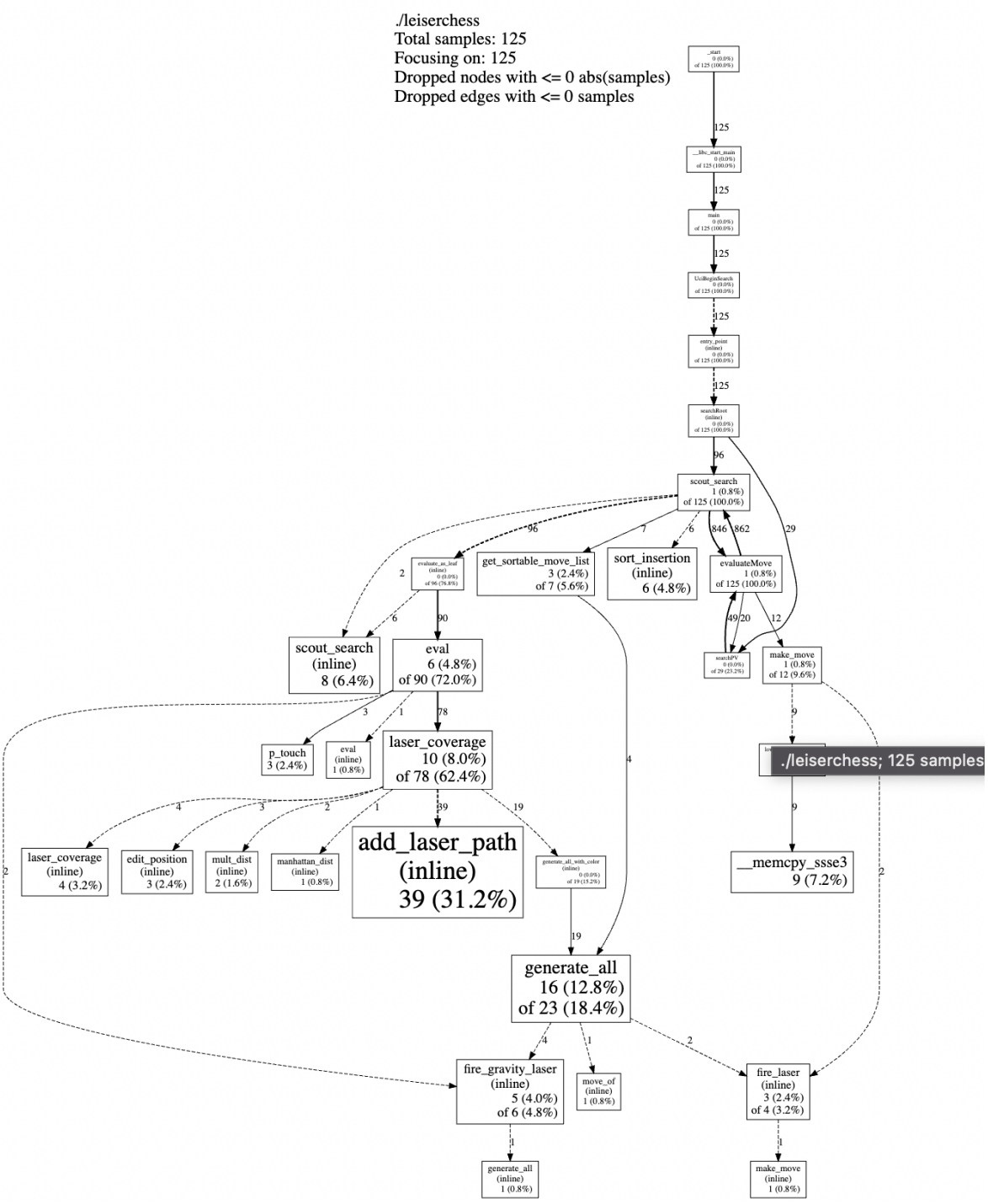

Figure 1: Diagram of Beta bottlenecks according to Google's pprof

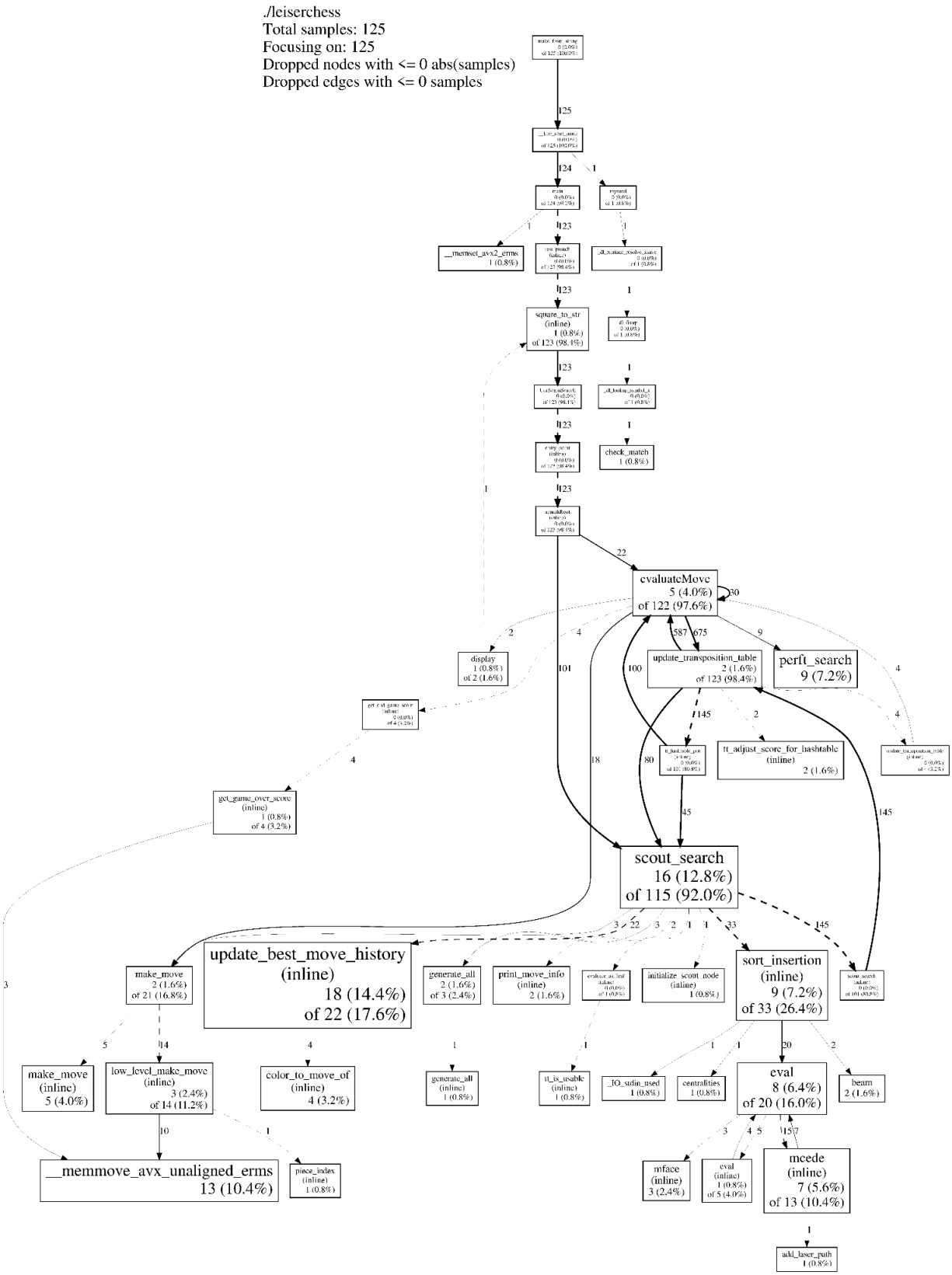

Figure 2: Diagram of Final bottlenecks according to Google's pprof