Project 4 Design Document

Design

How the different parts fit together (The Quantum Mechanic):
At each turn, the Leiserchess bot must choose the best move, using a variant of minimax search called Principal Variation Search (PVS) that is implemented in search.c. At the non-leaf nodes, the search function generates all possible moves (in move_gen.c) and proceeds recursively with the search. At leaf nodes, an evaluator function in eval.c is called that returns an integer score associated with the current position. During evaluation, computation of laser coverage requires increasing search depth by one additional ply to see which squares could be attacked by the laser next turn (and therefore where the enemy monarch cannot safely move).

Several tables are constructed to accelerate this search. In particular, a transposition table (in tt.c) serves as a cache of all positions analyzed in case this position is duplicated elsewhere in the search. This can occur if two moves can occur in multiple possible orderings, that is, if the moves can be transposed.

Preliminary Measurements and Initial Optimizations (The Quantum Mechanic):
In the initial codebase provided by the staff, a significant bottleneck was function calls to trivial, one-line functions, which collectively accounted for about 10% of total runtime. Enabling link-time optimization (through the compilation and linkage flag -flto) was a simple change to the Makefile that increased performance.

About a third of the runtime was spent in calls to the memory-copying function __memmove_avx_unaligned_erms, used to duplicate structs. This is called in low_level_make_move to copy the position before and after the move. For generating a branching tree for alpha-beta search, multiple copies of the position are needed, but for evaluating laser coverage in the leaves, it is not necessary to duplicate the board for a simple test move. For simple test moves in the evaluation of laser coverage, it is possible to edit the current copy of the position without duplication (and simply undo this at the end), which eliminates most of the copy operations and therefore about a third of total runtime.

The original laser coverage computation was also inefficient in that it tries every possible move from a position and computes the laser trajectory for that path, regardless of its impact on the trajectory. Many moves have no impact on the trajectory (e.g. they move a pawn from a square off the path to a different square off the path), so it is not necessary to consider all these uninteresting moves separately. Eliminating all but one of these “null” moves can reduce laser coverage runtime by about a third.

There were a couple other small changes made. A smaller but significant contribution to __memmove_avx_unaligned_erms was copying of a (mostly empty) list of moves returned from

evaluateMove(), which could be eliminated by passing the function result by reference rather than making it a return value. We also precomputed commonly used expressions (like inverses of small integers) in laser_coverage.

Altogether, this accelerated code performance by a factor of 2-3, and computing laser coverage is now only about half of total runtime rather than 90%.

After this initial pass, the following bottlenecks remain:

- According to perf, 55% of total runtime is in laser_coverage (within this: ~15% is making the move on the board and setting it up, ~50% in add_laser_path, ~25% is computing distances + doing divisions).
- add_laser_path is a substantial fraction of checking laser coverage - currently follows laser path one square at a time. Here, a significant amount of time (around 40% of add_laser_path) is spent deciding if the current laser path reaches squares in less distance than previously. This will be the most important thing to optimize, and there is probably some low-hanging fruit here that will be relatively quick to optimize.
- __memmove_avx_unaligned_erms, which is called to copy the board state from each node to its children, consumes around 5-10% of total clock cycles. Compressing the board representation (as detailed below) will accelerate this.
- 14% of total runtime is generating the list of moves.
- 8% in eval (heuristics), excluding laser coverage.

Changes to UCI Interface (Paint Splatter):
As an important tenet of project testing, we need a way to verify that internal data structures stay consistent and no internal invariants are violated during integration testing. Much of this task can be covered by compiling in sanitizers such as ASAN, TSAN, and MSAN, but sometimes a data structure has no memory corruption bugs or race conditions and is still in an incorrect state. User-defined internal invariant checking is needed for this case at runtime, and to this point no such framework exists in leiserchess. We contribute two additions to the UCI to remedy this issue: CHECK_REP and DUMP_DS.

CHECK_REP resembles the homonymous “check_rep” function used in 6.031, and grants users the ability to do custom invariant checking on data structures. Users can define a CHECK_REP for their data structures as follows, inside the same .c file where said data structure sees frequent use/is implemented:

```
CHECK_REP(my_data_structure_name){
  ...do invariant checking here
  return CHECK_REP_SUCCESS;
  /* return CHECK_REP_FAILURE; */
}
```

From here, a user can call SHOULD_CHECK_REP(my_data_structure_name) from anywhere in the project that includes consistencytest.h, an auto-generated header that implements CHECK_REP functionality. If the PEDANTIC flag is not set at compilation time (the default, more on that later), and the UCI option "check_rep" is set, SHOULD_CHECK_REP() will call the CHECK_REP. Otherwise, it will do nothing.

Additionally we introduce DUMP_DS, an enhanced manual debugging mechanism (read: glorified print statement) for failing checkreps and complex data structures. DUMP_DS allows users to print out their data structures to the UCI console. If a user wants this functionality, they can declare the following in the same .c file where their data structure is implemented (or, truth be told, pretty much any .c file in the project):

```
DUMP_DS(my_data_structure_name){
  ...printf data structure info here,
  ...anything here that you think you'd
  ...like to see while debugging
  /* printf("my_important_thing: %d\n",
my_global_data_structure->some_int);
}
```

Once this function is defined, a user can go to the UCI if the project is compiled without PEDANTIC set and type:

```
dump my_data_structure_name
```

...and the corresponding DUMP_DS function will be invoked, printing the user's data structure to the screen.

Note that we also introduce the PEDANTIC compile time flag. If PEDANTIC is set, both CHECK_REP and DUMP_DS will be completely disabled and completely compiled out of the leiserchess binary, and the UCI will conform strictly to the specification compiled by the staff (hence the phrasing choice). All submissions and perf testing will be done with PEDANTIC switched ON, even though the non-pedantic UCI additions should have minimal performance overhead.

Makefile Flags:
See **Preliminary Measurements** and **Initial Optimizations and Determinism**

Optimization Plans & Data:
See **Division of Labor** and **Preliminary Measurements and Initial Optimizations.** Tl;dr First items are board packing and making laser_coverage much more efficient, as well as testing well.

Determinism (Paint Splatter)

In some circumstances it is valuable to disable determinism in order to properly debug search mechanisms. We add a flag to the Makefile (DETERMINISM) which allows this, by manipulation of the options struct + a few other preprocessor flags scattered throughout the codebase. We've tested this thoroughly and verified that our definition of determinism aligns with the behavior of this flag.

Code evaluation metrics (ALL)
If the function is designed to be a drop-in replacement (i.e. pre-computing inverse distances between squares before runtime), unit tests should be written for this function and it should pass both consistency and reference tests. For changes that are not drop-in replacements for starter code, playtesting should also be used. New changes of this variety should beat old code by a statistically significant (2 sigma) margin.

Best-move (indiscreet_math)
This definitely seems like a 2nd-stage optimization. The basic idea is to keep a set of moves that are best by position and before computing the best move for a position, check against this table. Our plan is to start out in laser_coverage, and implement it that runtime down. One way we might choose to implement this is by keeping track of the best move for some *n* layers of the search.

Transposition Table (indiscreet_math & The Quantum Mechanic)
The transposition table (TT) stores the score of any position that has been seen (to avoid recomputation). This is something that's going to be somewhat hard to test because it breaks traditional correctness metrics. Our plan is to start off with tuning k to get an idea of the general trends, and retuning as we improve the code.

The hyperparameter *k* which could be tuned to potentially yield a simple improvement to TT queries. The table is a *k*-way associative table that uses hashing to insert elements into size-*k* buckets within which linear search is used. The bigger the buckets, the slower the table, but the more things can be stored before collisions become frequent. Intuitively, big *k* is good for high amounts of hash collisions, while small *k* is good for few collisions. Additionally, in the table, whenever there are *k* elements and one is inserted, one is evicted. Thus, we could either count the number of evictions and/or the number or hash collisions and try to tune it accordingly (i.e. smaller *k* is better if there are low collisions and/or low evictions).

Empirical testing with the current state of our code (on 1000 games, with slightly accelerated time controls) found that *k* = 1, 2, 4, and 8 were statistically indistinguishable, and *k* = 16 was slightly worse (with an Elo of -27 +/- 25 relative to the average of the five values studied). We will therefore stick with the default value for now. However, as the rest of the codebase improves and we increase the number of nodes searched, it may be advisable to increase associativity. Linear searches can be sped up with parallelization if necessary for larger sizes of *k*.

Killer-Move Table (Knows All 10 Digits of Pi):

This is a short circuiting table to avoid searching when there are killer moves. A killer move is a move that is really, really good and we want to immediately take it if the enemy doesn't somehow avoid it. It is likely a later stage optimization since killer moves are a small fraction of total moves. We are considering board representation optimizations below that will be effective for the killer move table (specifically for moves that kill the king, or potentially a pawn).

Eval Function (The Quantum Mechanic & Knows All 10 Digits of Pi):
Move generation happens at every non-leaf node in the move tree, while move evaluation occurs at all leaf nodes. Since the tree grows exponentially, leaf nodes account for the vast majority of nodes in the graph, so calls to the evaluation function account for the majority of computational work. The computation of laser coverage is particularly expensive since it requires searching an additional ply farther, so it is the primary target for optimizations. For the final submission, we will also explore parallelization of laser coverage since the various moves considered can be studied in any order.

Initially, our plan is to maintain the heuristic given by the staff for move evaluation. However, as we develop more knowledge of the game, it may be advisable to retune parameters affecting the various weights. Furthermore, there may be slight modifications that will enhance performance with little effect on gameplay, e.g., faster-to-compute approximations to the laser coverage function that agree in most cases. These will be worth exploring after exhausting the performance enhancements that can be made that preserve the initial results.

Moreover, we may consider using alternative strategies to the linear combination one which is currently used. For example, we may be able to save on runtime by using short circuiting of some kind to quickly eliminate bad positions, or we may be able to combine faster-to-compute approximations with randomness to be "good enough on average" and potentially enable computational savings.

Move Gen (The Quantum Mechanic):
The original code calls low_level_make_move in two cases:
1. Generation of the moves for the move tree that will be alpha-beta pruned
2. Extending the move try by an extra ply for checking laser coverage
The second case accounts for most calls to move generation in an exponentially growing tree, but the second case is also more specialized in that we only need to look at a subset of the moves, and we only need them temporarily (and will not need to branch from them). As detailed above, it is not necessary to store the full move -- we can just tweak the position and undo it later. Likely even the enumeration of moves can be specialized in this latter case. Therefore, most of our efforts thus far (and likely most of our future efforts) have been spent on the case where move generation is used for checking laser coverage.

Board Rep and Data Structures (indiscreet_math & Knows All 10 Digits of Pi):

Board

Currently, the board representation takes way too much space (each board position is 3896 bytes) and is copied repeatedly and iterated through in laser_coverage. The current board rep is also 16x16. Initial improvements would be to pack the board into a 10x10 format, then 8x8 format, then each square within that into 8 bits (3 bits for orientation, 2 bits for color, 2 bits for piece), meaning that the entire board fits in 8x64 bit ints. This representation would greatly speed up the memory copying (that is currently 5-10% of runtime) and could accelerate memory access as well.

More testing is needed to determine optimal packing for the enums and structs, because it may be more efficient to deal with 8-byte structs if they're being touched a lot. The board sizing and packing into 10x10 or 8x8 should bring significant improvements with respect to copying and laser coverage.

Data Structures

It may be advisable to use a graph-like data structure to encode the game. A graph data structure may be very good for a subset of the killer move table regarding winning moves, since it may be (1) more compressed than the default board (as we will soon see), (2) numerical and having different values per graph making it easier to hash, (3) it's independent of geometry, meaning that it captures rotational, translational, (etc) symmetries that maintain the topology, which reduces the storage size.

Note how simple each possible move is to the board graph. Moving the king either does nothing and may add or remove an edge. Killing a pawn removes a node and its edges (of both pawns, potentially). Rotating pawns destroys edges and may create new ones (as does rotating the king). Moving a pawn also destroys its edges and may create two new edges. Every move should be a node addition/delete, edge addition/deletion, or pair of such changes.

Graph Idea

The graph would store laser connectivity between pawns and the kings. Any two pawns have an edge between them if (1) there is nothing between them and (2a) they are on the same column and one is facing north and the other south, or (2b) they are on the same row and one is facing east and the other west. The king is connected to the first pawn it's facing.

We could store 16 shorts, each that encodes whether it is connected to each other element. (Note that there are 15 pawns and two kings: we could ignore one king and check connectivity with regards to the pawn it is connected to). Thus we need 16 * 16 = 256 elements which is 4 machine words. Then we can do updates easily by toggling bitmasks. We can detect whether two nodes are connected using DFS. The only difficulty is figuring out how to find new connections based on rank and file. We could improve this by having each element remember

whom it's connected to on each of its four sides since that is the maximum number of legal connections. Thus, we could use 4 bits to store the connection ID, have 4 of these for a total of 16 bits per element, then have 16 of these again. This would make it easier to see when a movement does not change the connectivity. However, it does not offer ideal symmetry and it is still hard to figure out how to detect new connections.

Union Find
Alternatively, we could use union find. We could use trees with path compression to union find across all the elements in the graph and see if the king is in the same set as the other king. We would iterate rank by rank and use a temporary variable to remember, for each file, where the previous pawn was and where it was facing. This way we could find connected pairs in linear time relative to the number of cells. The union find operations would be sub-logarithmic in the number of cells.

Why We Will Avoid Graph Data Structures For Now
Now, while all this is well and good, we think that there are much more important bottlenecks to optimize. Right now, laser coverage is our main bottleneck. However, if the laser coverage can be accelerated, then other bottlenecks such as this should be explored.

Laser Path Cache
Instead of using a graph, we are also thinking of storing a copy of the board which stores at each location either zero, or a value pertaining to the sum (up to that point) of the distance metrics of the cells on which the laser (for some color) is traveling (with respect to both its own king and/or the enemy king). Whenever we explore the laser coverage for a child, we see if the move that took us to that child led to a change one one of the cells on the path. For that single cell and those after we re-calculate the sum, but not for those before. This should, on average, save us around half (assuming a uniform distribution) of the laser coverage computation.

Testing Framework (Paint Splatter)

A project of the scale of Leiserchess requires a robust testing framework, and the stock project four codebase lacks any sort of testing infrastructure beyond play testing and a perft framework to perform basic 'smoke tests', i.e. regression tests for move generation. We'd like to improve this situation by introducing four forms of tests:

Unit tests: Good unit tests should be easy to invoke throughout the codebase, offer a clean-slate environment capable of catching signals as well as test failures, and disappear in production builds. To this end, we introduce unittest.h, a simplistic unit testing framework that accomplishes all of these tasks. This framework has been implemented and a PR is currently open to see it merged into our mainline.

Unit tests can be declared in any .c file using the following syntax:

```
UNIT_TEST_FN (my_unit_test_name) {
```

```
  ...do your unit test stuff...
  return UNIT_TEST_SUCCESS // or UNIT_TEST_FAILURE
}
```

Running unit tests is as simple as recompiling leiserchess with the UNIT_TEST flag set. This sort of build hooks over the leiserchess main runloop and invokes all unit tests instead, printing out information about whether the tests passed, failed, or crashed. Some sample unit tests that use this syntax are in our util.c.

Under the hood, the unittest-aware build system invokes a script called unittest_gen.pl before compiling any object files. If the UNIT_TEST flag is set the script will scan the source tree looking for tests and construct unittest.h, which contains function pointers to all unit tests along with their names in string form. Otherwise, the script generates a unittest.h with a UNIT_TEST_FN macro that ensures that unit tests are optimized out - this functionality has been intensively verified.

A nice feature of our unit tester is that it executes all tests in their own address space. This ensures we can catch memory corruption bugs (e.g. segmentation faults or floating point errors) without forcibly terminating the tester. This makes unittest.h suitable for deployment in automated environments, which we might consider adopting as progress on the project grows more multifaceted.

Consistency tests: Consistency checks play the bot against itself using a variety of compiler flags and added UCI features, to ensure internal invariants remain steady throughout all facets of program execution and that no memory corruption or data races occur. Consistency checks always enable the CHECK_REP framework and run the binary against itself a number of times under the following compile-time conditions:

- Compiled under ASAN
- Compiled under TSAN
- Compiled under MSAN (undefined read sanitizer)
- Compiled normally, attached to an additional harness to monitor memory footprint and/or CPU usage statistics

Should a sanitizer fail, the consistency test will also fail. Moreover, a user should be able to set thresholds for memory footprint and trigger a test failure should these thresholds be surpassed.

Reference tests: Reference tests disable non-determinism in the leiserchess binary, allowing a deterministic framework to run known searches, static evaluations and ensure outputs match a known working binary. These primarily test the search routines in the absence of any tables as an effective starting point for parallel work. Perft output also falls under this category, since it can be analyzed fully without non-determinism from fixed, user-specified board configurations. A script in our repository, test.sh, simulates this functionality, and a more complete solution will come prior to beta submission.

Play tests: These tests pit two competing implementations against each other, enable PEDANTIC mode on the respective binaries, and rate ELO over a set of [number of CPUs on your machine] runs conducted in parallel. This is our last line of defence against errors not spotted by reference tests and consistency tests -- the objective is not to use this for debugging (together, reference tests and consistency tests should accomplish our integration testing needs) but to merely use this as a thin wrapper around the autotester. This is also useful for changes that affect functionality rather than just performance, such as tweaking the evaluation heuristic.

Thoughts on Parallelism:

The search tree seems like a great place to start with respect to parallelism, as the sides of the tree are non-overlapping, and most of the work is being done in this part. (This is a little tricky, though, since many nodes are ultimately rejected during alpha-beta pruning.) Also, evaluating all of the laser_coverage would be a place to throw some parallelism in for similar reasons, and since laser_coverage is computed within a leaf, this avoids any difficulties with ordering the evaluation of nodes. That being said, we haven't really made a dent in the project yet, so it's entirely plausible and even likely that we'll have to put parallelism in different places.

Division of Labor:

This represents our initial projection for division of labor and will change over time with the flow of the project.

- **Paint Splatter:**
    - Implement testing mechanisms
    - Develop comprehensive tests/invariant checks for all significant data structures, with priority set by the flow of optimization on the project
    - Assist with other folks on optimization on demand
- **Indiscreet_math**:
    - Board rep packing
    - laser_coverage optimization
- **The Quantum Mechanic:**
    - Start by optimizing laser_coverage (and especially add_laser_path)
    - Then profile to look for subsequent optimizations
    - Investigate construction of an endgame table and perhaps an opening book
- **Knows All 10 Digits of Pi:**
    - laser_coverage optimization
    - Explore graph data structures
    - Look at endgame table