Technical summary for Firefly chess engine

1. Board representation (src/chess/board.h/.cpp)

The board is represented by four packed uint64_t fields

```
uint64_t current_player_pieces;
uint64_t bishops_queens_kings;
uint64_t rooks_queens_knights;
uint64_t pawns_knights_kings;
```

along with 16 bits of flags for special rules

```
bool flipped:1;
uint8_t en_passant_x : 3;
bool en_passant_possible : 1;
uint8_t castling_rights : 4;
uint8_t halfmove_clock:6;
bool has_repeated:1;
```

Recovering the positions for the individual pieces is done through a series of set intersections and differences.

```
kings   = bishops_queens_kings & pawns_knights_kings;
queens = rooks_queens_knights & bishops_queens_kings;
knights = pawns_knights_kings & rooks_queens_knights;

bishops  = set_difference(bishops_queens_kings, (queens | kings));
rooks    = set_difference(rooks_queens_knights, (queens | knights));
pawns   = set_difference(pawns_knights_kings, (knights | kings));
```

ANDing them with current_player_pieces will produce the positions of each piece for the current player, black if flipped, white otherwise.
Conversely, a set_difference will produce the positions of each piece for the opponent.

The final board is 34 bytes (compiler pads it to 40), a desirable optimization, as each node contains a board.

2. Memory management (src/mcts/memory.h/.cpp)

The memory manager allocates large chunks of memory (8MiB by default) and partitions them into nodes as needed.
Nodes are fused with their edges, making the start of the edge list from the perspective of a node simply *this + 1*.

Deallocation is done in bulk, typically after a move was made by the opponent, the deallocation itself is more of a reallocation, all nodes from the current subtree are parsed and moved from their current positions in memory to the beginning of all memory, eliminating any gaps between them, old roots are saved externally, for use as history planes.

This should sidestep any heap fragmentation issues, while providing good cache-locality and removing the need for iterating through all nodes to be deallocated, though an unfortunate side-effect is that destructors don’t get called, so memory allocated from outside the memory manager will be leaked, though such allocations should never be necessary.

2.1. Transposition tables

Transposition tables are also handled by the memory manager, currently they're implemented as simply

```
std::unordered_map<uint64_t, mcts::node*> transposition_table;
```

On a memory free operation, the table is cleared and reallocated nodes are reinserted.
The key is a hash acquired by hashing the board representation with xxHash.

3. Nodes and Edges (src/mcts/node.h/.cpp)

The basic elements of a tree, each node represents a position, each edge represents a move that can be made from that position.

Each edge is a 16 byte structure that looks something like this:

```
struct edge {
   /*
    * If not expanded node == nullptr
    */
   node *node_;


   union {
      float P_;
      float terminal_value;
   };

   chess::move move;
   bool terminal;

...
```

Edges which lead to terminal states do not get expanded, instead they are simply marked as *terminal* and a *terminal_value* is set.

If an edge does not lead to a terminal state, then *P_* contains the prior probability that *move* is made, as calculated by some evaluator, if the edge is already expanded *node_* will contain the address to the resulting node.

As already stated, edge lists are contiguous with their parent node, edge lists are volatile, edges may be reordered by any thread at any time, if a pointer to an edge is held, the lock to the parent node must also be held (see 3.1).

Node representation:

```
class node {
...
node *parent;

chess::board board;

floatx Q_;

uint32_t visit_count;
copyable_atomic<uint32_t> visits_pending;


uint8_t edge_count;
uint8_t index_in_parent;
```

```
uint8_t viable_edges;

uint8_t moves_left; // moves_left == min(moves_left, 255)

uint8_t repetitions: 2;
bool reversible_move: 1;
bool evaluated:1;
solution_state solution : 2;
```

First off *Q_* isn't actually a Q value as obtained from a neural network, or rather it is only when *visit_count* == 1, later it becomes an average value for the whole subtree, but the name kind of stuck.

viable_edges and *solution* are used for node solving (see 6.)

3.1. Thread Synchronization

```
class node {
...
static thread_local uint8_t thread_id;
copyable_atomic<uint8_t> locking_tid;
uint8_t lock_count;

inline void lock()
{
   if (locking_tid == thread_id) {
      lock_count++;
      return;
   }
   uint8_t f = -1;
   while (!locking_tid.compare_exchange_weak(f, thread_id)) {
      f = -1;
      std::this_thread::yield();
   }

   lock_count = 1;
}

inline void unlock()
{
#ifdef DEBUG_CHECKS
   if (lock_count == 0)
      throw std::logic_error("Unlocking non-locked node.");
#endif

   if (--lock_count == 0)
      locking_tid = -1;
}



...
```

Each node can act like a *std::recursive_mutex*, though to make this work, every thread needs to initialize the thread_local *thread_id* before any tree traversal.

This allows for relatively cheap and simple mutual exclusion, while preventing any deadlocks from occurring within the same node.

There are some caveats to this solution, consecutive nodes cannot be locked on both forward and backward tree traversal, as it's possible for two threads to "meet" and try to lock each others nodes, producing an unrecoverable deadlock. Currently, backward traversal can lock consecutive nodes, while forward traversal is restricted to locking single nodes.

Once the traversal algorithms are in a more stable state, the recursive mutex functionality may be removed in favor of a simpler and cheaper one time locking and unlocking mechanism.

There are separate synchronization mechanisms for things like transposition table probing, memory allocation, inference, etc.

3.2. Turning the tree into a graph (Future)

This seems to be the most reasonable way to handle transpositions, however a simple multiparent approach clearly won't work, as it's impossible to tell which parent to traverse for history planes and threefold repetitions. Thus, it seems that the only viable solution is to store the path to each leaf externally and use that as instead of parents.
However, backpropagating Q values through all parents also seems like a reasonable idea, so a hybrid approach may be worth considering.

4. Neural network (src/neural/lc0_network.h/.cpp)

Currently, the engine depends on and is only compatible with neural nets from the Leela Chess Zero project.
Topology
Training nets specifically for Firefly is a future goal, though the goals outlined in (6.) take precedence as they are likely to influence the training process.

4.1. Network manager (src/neural/network_manager.h)

Does input processing, inference (possibly on multiple devices) and output processing.

Originally, it was an asynchronous pipeline, with the idea of reducing latency and increasing batch sizes, this ended up being a bad idea as its very difficult to get it to scale.

Currently, it's a synchronous algorithm in which every thread selects nodes and adds them to a shared batch, any thread which hits an expanded but unevaluated node is responsible for processing the batch.
This works well enough for now.

5. Search (src/mcts/search.h/.cpp)

This is the intersection of all the pieces of the engine, it handles threading, listens for search termination conditions and implements the central PUCT selection/evaluation loop.

6. Solving positions

A position may naturally become solved as terminal edges are found.
A node's viable_edges field signifies the number of edges which do not lead to a solved or terminal position, when this value reaches 0, the node is solved, however this is not necessary.
As an example, a node which leads to at least one checkmate becomes solved as losing no matter how many viable_edges it has left.

If a position does become solved, then $Q\_ \in \{-1,0,1\}$

the difference between the approximate Q_ and the solved Q_ is the delta, multiplied by the node's visit_count, it becomes a weighted delta, which is then backpropagated up the tree, improving the accuracy of the branch.

For reference:
(mcts::edge::set_terminal, mcts::node::adjust_value_for_solved_branch, mcts::node::propagate_solved_value)

```
inline void propagate_solved_value(float weighted_delta)
{
   lock();
   auto new_value = (Q_ * visit_count + weighted_delta) / visit_count;

   weighted_delta = (Q_ - new_value) * visit_count;
   Q_ = new_value;
   unlock();

   if (parent && parent != current_root)
      parent->propagate_solved_value(weighted_delta);
}
```

Further elaborating on this system is a future goal, it seems possible to incorporate minimax searches to efficiently solve nodes close to many terminal states, which should be beneficial particularly when there's a lot of uncertainty in the node value, which may be quantified as something like $0.7 > |Q_| > 0.3$.