LECTURE 19
Leiserchess Codewalk
Helen Xu
November 15, 2018
GAME RULES
Leiserchess Board Game

Two players: Tangerine & Lavender

Each player has 7 Pawns and 1 King

- Pawn: 4 Orientations
- King

© 2008–2018 by the MIT 6.172 Lecturers
General Gameplay

- Tangerine moves first, then play alternates between the two players.
- All pieces move the same (King or Pawn)
- Each turn has two parts: moving and firing the laser.
- The laser reflects off the long edge of the pawns and kills a pawn if it hits the other sides.
- One side wins when its King shoots the other King with its laser.
How to Move

• At the beginning of each turn, the player on move chooses a piece to move.
• There are two types of moves: basic and swap.
Basic Moves

On a *basic* move, a piece can either:

- rotate 90, 180, or 270 degrees
- move to an empty adjacent square in any of the eight compass directions while maintaining orientation.
- A piece cannot both rotate and move.
If an enemy piece occupies an adjacent square to the player’s piece, the two pieces swap squares (maintaining their orientation) and the player’s piece must make an extra basic move.

Initial position with Tangerine to move.  
Intermediate position after swap.  
Final position after extra basic move.
A Ko rule (familiar from the game of Go) helps to ensure that the game makes progress.

A move is illegal if it “undoes” the opponent’s most recent move by returning to the position immediately prior to the current position.

Tangerine performs a swap move. Lavender performs a swap move to undo Tangerine’s move.
Draws

• A draw occurs if:
  • There have been 50 moves by each side without a Pawn being zapped,
  • The same position repeats itself with the same side on move, or
  • The two players agree to a draw.
A chess clock limits the amount of time players have to make a move.

When it’s your move, your clock counts down.

When it’s your opponent’s move, your clock stops.

We shall use Fischer time control, which specifies an initial time budget and a time increment.

The notation \texttt{fis 60 0.5} means each player is allocated a time budget of 60 seconds to begin, and 0.5 seconds is added to the budget each time the player makes a move.

https://en.wikipedia.org/wiki/Time_control

© 2008–2018 by the MIT 6.172 Lecturers

Image is in the public domain via the Dutch National Archives.
For a King to zap the enemy King, it risks opening itself up to counterattack.

For example, how can Tangerine zap the Lavender Pawn on \texttt{f5}?
Tangerine can zap Lavender’s pawn on f5 by moving its pawn on c2 to c1. Now, how can Lavender counter?
Lavender can counter by moving its pawn on d6 to c7, zapping the Tangerine King and winning the game.
Forsyth–Edwards Notation (FEN)

FEN describes a chess position using a character string (see player/fen.c).

Example (opening position):

```
ss7/3nwse3/2nwse4/1nwse3NW1/1se3NWSE1/4NWSE2/3NWSE3/7NN W
```

[Image: Graph showing chess board notation]

David Forsyth

[Link: https://www.chessprogramming.org/Forsyth-Edwards_Notation]

© 2008–2018 by the MIT 6.172 Lecturers

This image is in the public domain.
FEN describes a chess position using a character string (see `player/fen.c`).

Example (opening position):
```
ss7/3nws3/2nws4/1nws3NW1/1se3NWSE1/4NWSE2/3NWSE3/7NN W
```

Slashes separate rows
- 1 space
- Lavender Pawn facing SE
- 3 spaces
- Tangerine Pawn facing NW
- Tangerine Pawn facing SE
- 1 space

[https://www.chessprogramming.org/Forsyth-Edwards_Notation](https://www.chessprogramming.org/Forsyth-Edwards_Notation)

© 2008–2018 by the MIT 6.172 Lecturers
Forsyth–Edwards Notation (FEN)

FEN describes a chess position using a character string (see `player/fen.c`).

Example (opening position):

```
ss7/3nwse3/2nwse4/1nwse3NW1/1se3NWSE1/4NWSE2/3NWSE3/7NN W
```

Player to move.

- W = Tangerine
- B = Lavender

[Link](https://www.chessprogramming.org/Forsyth-Edwards_Notation)
Algebraic Notation for Games

1. g4g5  e6e7  19. f3f5  c7c6  35. e5e6  c3c2
2. g5g4  e7e6  20. e4d3c4  a7b7  36. a1b1  c1L
3. g4L  b3L  21. e4d3c4  a7b7  37. b1R  d0R
4. g4h5  b4R  21. a5L  d7d2  38. e0L  c1b2
5. f3e4  d5e4f5  22. h4h2  d1c0  39. b1b2c1  b1a1
6. f2e3  f5g5  23. e4f5e5  e4f5  40. c1b2  a1b2a3
7. h5g6  g5h4  24. c2f3e3  h7c7  41. e6U  a3R
8. h0g0  c4R  25. b5b2  e5e1  42. e6d5  a7R
9. e3e4  b3b2  26. b4b1  c6e6  43. f5e6  c2U
10. g0f0  d6d7  27. g5f6e5  a7L  44. a1b1  c7U
11. d5c5b5  h4g3f2  28. g6f6e5  d7d2  45. e0f0  c2d2
12. e2f2g2  c4b3  29. g6f5e5  e4c3  46. b1c1  c7b7
13. e4d5R  d7L  30. e1e0  d4c3  47. d5L  d2e3
14. d5e6e7  b2b1  31. f5g6f6  c3c2  48. f0g1  e3f3
15. e1d0  e4e1  32. d5d3c3  2d1c0  49. g1h1  f3g3
16. b5c4  d5d5e4  33. d5d3c3  0-1  50. e6f5  g3h3
17. d5R  e5e6  0-1

1–0 Tangerine wins
0–1 Lavender wins
1/2–2/1 Draw

https://en.wikipedia.org/wiki/Algebraic_notation_(chess)
PROJECT ORGANIZATION
Directories under project4:

doc: Leiserchess rules and documentation for the game–engine interface.

autotester: Java local autotester

BayesElo: parses Elo results from autotester.

pgnstats: parses statistics from autotester results.

tests: test specifications for the local autotester.

player: code for your Leiserchess bot. You will be optimizing the code in here!

webgui: local webgui where you can watch the game and play it.
The Java local autotester is in `autotester/` under the code distribution.

You can test changes to your bot using time trials over many games.

The `tests/` directory holds configuration files for your autotests:

- number of games,
- bots in your trials,
- time control,
- etc.
cpus = 12  
book = ../tests/book.dta  
game_rounds = 500  
title = basic

# now we have the player definitions
# --

player = reference  
invoke = ../player/leiserchess  
fis = 20 0.5

player = with_change  
invoke = ../player/leiserchess_with_change  
fis = 20 0.5

Modified from tests/basic.txt.

© 2008–2018 by the MIT 6.172 Lecturers
Leiserchess uses the Universal Chess Interface (UCI), a communication protocol for automatic games to pass information between the bots and the autotester. UCI allows the programmer (or autotester) to enter the move made by the game engine.

https://www.chessprogramming.org/UCI
Elo Ratings

The Elo rating system measures relative skill levels in zero-sum games like chess.

A player’s Elo rating depends on the Elo ratings of its opponents.

Example output from autotests:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Elo</th>
<th>+</th>
<th>-</th>
<th>games</th>
<th>score</th>
<th>oppo. draws</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>test6</td>
<td>269</td>
<td>137</td>
<td>100</td>
<td>33</td>
<td>94%</td>
<td>-140</td>
</tr>
<tr>
<td>2</td>
<td>test5</td>
<td>40</td>
<td>96</td>
<td>98</td>
<td>33</td>
<td>55%</td>
<td>-29</td>
</tr>
<tr>
<td>3</td>
<td>test4</td>
<td>-309</td>
<td>113</td>
<td>185</td>
<td>34</td>
<td>3%</td>
<td>155</td>
</tr>
</tbody>
</table>
The local webgui lets you watch a game — or even play one — without sending it to the scrimmage server.

You can run it using the commands in webgui/README.
MOVE GENERATION
(move_gen.c)
Any chess program needs a board representation to keep track of where the pieces are.

The reference implementation uses a 16x16 board with sentinels to store an 8x8 board.

https://www.chessprogramming.org/Board_Representation
https://www.chessprogramming.org/Mailbox
https://www.chessprogramming.org/10x12.Board
The position in the Leiservchess player stores the board representation, history, and other information about how we got to this point in the game.
Move Representation

- Piece Type (Empty, Pawn, King, Invalid)
- Orientation (4 choices)
- From square
- Intermediate square
- To square

28 bits
At each turn, our program needs to see all the moves it can possibly make.

In `move_gen.c:286`, we generate all the moves given a position depending on whose turn it is.

In the reference implementation, we iterate through the entire board and generate all the moves for each piece of the right color when we pass by it.
Perft is a debugging function that enumerates all legal moves of a certain depth (move_gen.c:698).

If you modify the move generator, make sure that Perft returns the same results.

```c
uint64_t Perft(int depth) {
    move_t move_list[256];
    int n_moves, i;
    uint64_t nodes = 0;

    if (depth == 0) return 1;

    n_moves = move_gen(move_list);
    for (i = 0; i < n_moves; i++) {
        make_move(move_list[i]);
        nodes += Perft(depth - 1);
        unmake_move(move_list[i]);
    }
    return nodes;
}
```

https://www.chessprogramming.org/Perft
MOVE ORDERING
Alpha-beta and principal variation search depend on putting the best moves at the front to trigger an early cutoff.

How do we determine which moves are best without static evaluation at every level?

We call `get_sortable_move_list` at `search.c:144` and implement it at `search_common.c:402`. 
Move Representation

Moves are represented in 28 bits (int32_t). If we want to make them sortable, we use 64 bits (int64_t) and use the upper 32 as the sort key.

The move representation is defined in move_gen.h:119.
STATIC EVALUATION (eval.c)
We use static evaluation to determine which positions are better than others (and therefore which moves we should make).

The function `eval(position_t* p)`, located at `eval.c:438`, generates a score given a position based on heuristics (higher means better).

At first, we suggest focusing on optimizing the existing structs and evaluation heuristics before coming up with new ones.
King Heuristics

• **KFACE**: bonus for your King facing the enemy king.

• **KAGGRESSIVE**: bonus for the King with more space behind it (to the end of the board)

• **MOBILITY**: how many spaces around your King are free.
Pawn Heuristics

• **PCENTRAL**: bonus for Pawns near the center of the board.
• **PBETWEEN**: bonus for Pawns between the two Kings.
**LCOVERAGE**: measures how much the board near the enemy king is covered by lasers after making all possible moves from a position.
ALGORITHMS FOR GAME–TREE SEARCH
Game Search Trees

Move generation (move_gen.c) to enumerate all possible moves from a position

Implemented in search.c

Move generation (move_gen.c) to enumerate all possible moves from a position

Position p

Move m

Position p’ after move m

Static evaluation (eval.c)

Depth d

© 2008–2018 by the MIT 6.172 Lecturers
Quiescence Search

- Evaluating at a fixed depth can leave a board position in the middle of a capture exchange.
- At a “leaf” node, continue the search using only captures — quiet the position.
- Each side has the option of “standing pat.”
- Implemented at `search_common:182`.

[Source](https://www.chessprogramming.org/Quiescence_Search#Standing_Pat)
Higher Depth Search = Better AI

Elo

Depth searched

© 2008–2018 by the MIT 6.172 Lecturers
Min–Max Search

- Two players: MAX ■ and MIN ●.
- The game tree represents all moves from the current position within a given search ply (depth).
- At leaves, apply a static evaluation function.
- MAX chooses the maximum score among its children.
- MIN chooses the minimum score among its children.
Alpha–Beta Strategy

- Each search from a node employs a window \([\alpha, \beta]\).
- If the value of the search falls below \(\alpha\), keep searching.
- If the value of the search falls between \(\alpha\) and \(\beta\), then increase \(\alpha\) and keep searching.
- If the value of the search falls above \(\beta\), generate a beta cutoff and return.
**IDEA:** If **MAX** discovers a move so good that **MIN** would never allow that position, **MAX**’s other children need not be searched — **beta cutoff**.
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
**IDEA:** If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
**IDEA:** If **MAX** discovers a move so good that **MIN** would never allow that position, **MAX**’s other children need not be searched — **beta cutoff**.
IDEA: If MAX discovers a move so good that MIN would never allow that position, MAX’s other children need not be searched — beta cutoff.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let's consider the pruning performed by Alpha-Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Alpha–Beta Pruning

Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.
Let’s consider the pruning performed by Alpha–Beta on a game tree that does not have an optimal move ordering.

11 Leaves Pruned
Theorem [KM75]. For a game tree with branching factor $b$ and depth $d$, an alpha–beta search with moves searched in best–first order examines exactly $b^{d/2} + b^{d/2} - 1$ nodes at ply $d$. ■

The naive algorithm examines $b^d$ nodes at ply $d$. For the same work, the search depth is effectively doubled. For the same depth, the work is square–rooted.
int search( searchNode* node, int depth ) {
    move_list = get_moves(node);  // struct searchNode{
    int score = eval(node->position)   //   searchNode* parent;
    // position_t position;
    if (abs(sc) >= MATE || depth <= 0){  // score_t alpha;
        // Leaf node
        // score_t beta;
        return score;  // bool abort;
    }  // score_t best_score;
    // }  
    // Negascout
    node->alpha = -node->parent->beta;
    node->beta = -node->parent->alpha;
    // Create child node for searches.
    searchNode child;
    child.parent = node;
Code for Alpha–Beta Pruning

16 // Generate moves, hopefully in best-first order
17 move_list = gen_moves(node->position);
18
19   for (mv in move_list) {
20     make_move(child.position, mv);
21     // Search the child
22     child_score = -search(&child, depth-1);
23
24     if (child_score > node->best_score)
25       node->best_score = child_score;
26     if (child_score >= node->beta) /* beta cutoff */
27       node->abort = true;
28       break; // Early Exit!
29     if (child_score >= node->alpha)
20       node->alpha = child_score
31   }
32 return node->best_score;
Principal Variation Search Pruning

Idea: Assume the first move is the best, and run scout search ("zero window" search) on the remaining moves to verify that they are worse.
Fail-Bad: If the zero window search returns a worse score than the first subtree, we can safely skip the full-window search in those subtrees.

**Subtrees executed with scout search**
Principal Variation Search Pruning

Subtrees executed with scout search
Let’s see a case where the scout search fails—good.

Full Window
Score in \([-\infty, 3]\)

Zero–Window Search
(from min’s perspective)
Score in \([3, 3]\)

Subtrees executed with scout search
Let’s see a case where the scout search fails—good.
Fail–Good: Zero-window search says the move might be better. Must do a full window search.
Principal Variation Search Pruning

Subtrees executed with scout search

© 2008–2018 by the MIT 6.172 Lecturers
Principal Variation Search Pruning

Subtrees executed with scout search
Scout search can improve pruning (modestly). Notice that most of the game–tree was processed using only zero–window searches...

13 Leaves Pruned

Subtrees executed with scout search
SEARCH OPTIMIZATIONS
Chess programs often encounter the same positions repeatedly during their search. A transposition table stores results of previous searches in a hash table to avoid unnecessary work.

- Call to update: `search.c:195`.
- Update function: `search_globals.c:56`.
- Used to order moves in `search.c:105`.

[https://www.chessprogramming.org/Transposition_Table](https://www.chessprogramming.org/Transposition_Table)
Zobrist Hashing

Zobrist hashing is a rolling hashing technique to convert a board position into a number of fixed length with uniform probability over all possible numbers (move_gen.c:112). The transposition table uses Zobrist hashing to index into it.

Note: If you change the piece representation and want to use node counts to debug, you must recompute the zobrist hash from the old piece representation.

https://www.chessprogramming.org/Zobrist_Hashing
The killer move table stores moves so good that the opponent would prevent you from going down that path, so you can early exit and avoid exploring that subtree.

The table is indexed by ply, because you tend to see the same moves at the same depth.

- Table at `search_globals.c:11`.
- Set at `search_common.c:378`.
- Used in `search_common.c:409`.

https://www.chessprogramming.org/Killer_Heuristic

© 2008–2018 by the MIT 6.172 Lecturers
The best move is stored at the root of a search and is the move that gained the maximum score.

The best-move table is indexed by color, piece, square, and orientation.

- Best-move history table at `search_globals:17`.
- Updated at `search_common:367`.

https://www.chessprogramming.org/Best_Move
Null–Move Pruning

Null–move pruning first tries to reduce the search space by not moving and then doing a shallower search to see if the subtree can still cause a beta cutoff.

If the tree can cause a beta cutoff even without a move, it is too good. The opponent would not let us go there, and so the search does not bother to explore it.

- Implemented at search_common.c:193.

https://www.chessprogramming.org/Enhanced_Forward_Pruning
https://www.chessprogramming.org/Null_Move_Pruning

© 2008–2018 by the MIT 6.172 Lecturers
Futility pruning only explores moves that have the potential to increase \textit{alpha}. It calculates this possibility by adding a futility margin (the largest possible gain) to the evaluation of the current position. If the result does not exceed \textit{alpha}, skip the search of this move.

- Implemented at \texttt{search\_common.c:209}.

\url{https://www.chessprogramming.org/Futility_Pruning}
Late–Move Reduction

After ordering the moves from a position, the moves at the front of the list are more likely to cause a cutoff.

Late–move reduction searches the first few (3 or 4) moves to full depth and the remaining ones with less depth.

- Implemented in scout search at `search_common.c:289`.

[https://www.chessprogramming.org/Late_Move_Reductions](https://www.chessprogramming.org/Late_Move_Reductions)

© 2008–2018 by the MIT 6.172 Lecturers
Opening books store positions at the beginning of the game.

Idea: Precompute the best moves at the beginning of the game.

They save time in searching and can store results to a higher depth.

The [KM75] theorem implies it is cheaper to keep separate opening books for each side than one opening book for both.

[https://www.chessprogramming.org/Opening_Book](https://www.chessprogramming.org/Opening_Book)

© 2008–2018 by the MIT 6.172 Lecturers
An endgame database is a table for guiding a chess program through the endgame.

For endgame positions, the distance from the end might be too far to search. With an endgame database, you can store who will win and how far you are from the end of the game.

`player/end_game.c` is a great place to store an endgame database.

[https://www.chessprogramming.org/Endgame](https://www.chessprogramming.org/Endgame)
Tips And Tricks
The Chess Programming wiki ([https://www.chessprogramming.org](https://www.chessprogramming.org)) is an invaluable resource for learning about the parts of a chess-playing program.
General Guidelines

- Test often! It is easy to make a mistake with your optimizations that does not appear when you just search to fixed depth.
- Testing methodology
  - WebGUI
  - Java Autotester
  - Cloud Autotester
  - Node counts
  - Function comparison testing
Optimization Tips

• Start with optimizations that do not affect the search (e.g. modifying the board representation).
• Improve the existing heuristics before trying to come up with your own.
• There are plenty of serial optimizations that you can make before thinking about parallelization.
6.172 Performance Engineering of Software Systems
Fall 2018

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms.