ECE469 – Checkers AI

Jonathan Lam

November 5, 2020

Since the assignment is fairly well-defined, this "report" will mostly be commentary on non-normative details. The relevant source code and toy states are located at `https://github.com/jlam55555/checkers-ai`.

1 Ruleset

- Ruleset from the Standard Laws of Checkers [1]. Jumps must be taken. Black moves first, white second. (In the C++ code I call the second player RED, but this naming difference is inconsequential.)

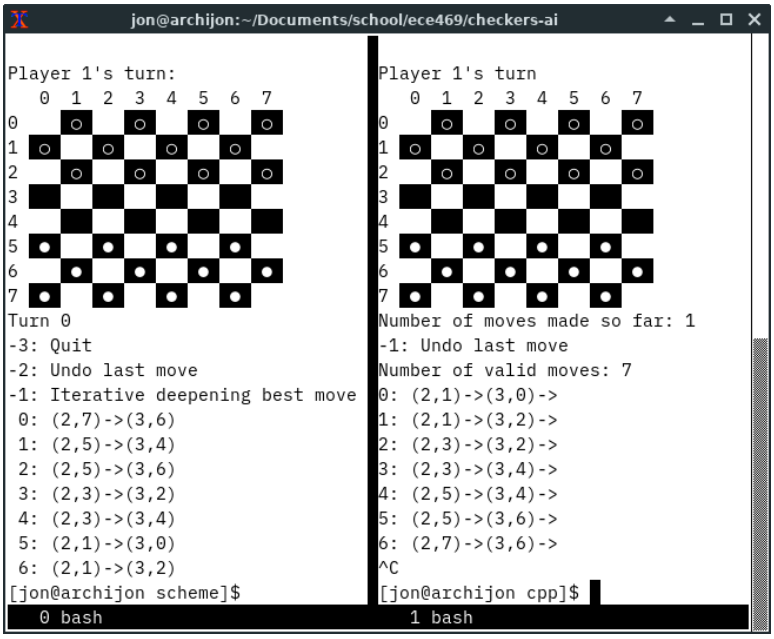

Figure 1: Comparison of TUIs: on the left is the Scheme version, on the right the C++ version. The differences are unintended and inconsequential.

2 Dependencies, compilation, and execution

The development environment was Arch Linux with g++ 10.2.0, Chez Scheme version 9.5.4. The game is played in a standard ANSI-compliant terminal with a monospace font.

2.1 Dependencies

The C++ version requires any modern version of g++. (clang/LLVM was not tested.)

The Scheme version requires that Chez Scheme is installed and the program `scheme` is in your path. Depending on the installation of Scheme, you may have to symlink the default scheme executable; e.g., if scheme is installed at `/usr/bin/chezscheme9.5`, then you will have to symlink it to the scheme executable (`ln -sf /usr/bin/chezscheme9.5 /usr/bin/scheme`) or change the interpreter on the shebang of `play-game.ss`.

2.2 Compiling and running the program

There's not really a need for a Makefile in either case. Navigate to the respective source directories and run:

2.2.1 C++

```
$ g++ -o checkers checkers.cpp
$ ./checkers
```

2.2.2 Scheme

```
$ chmod +x play-game.ss
$ ./play-game.ss
```

2.2.3 Scheme (REPL)

The Scheme version may also be played interactively through the REPL. Example commands are:

```
$ scheme                          # enter REPL
> (load "main.ss")                ; load game files

> (play-game)                     ; start interactive game

> (define state                   ; manually load state
    (car
      (load-state-file
        "../states/s1.txt")))

> (print-board state)             ; print board of current state

> (valid-moves state)             ; get valid-moves for state
```

```
> (list-ref                       ; print board after
    (valid-moves state) 3)        ; applying fourth valid move

> (print-move "First␣move"        ; print the first valid move
    (car (valid-moves state)))    ; (car gets first move)

> (define best-move               ; get best move from iterative
    (iterative-deepening-search   ; deepening search with a
     state 2))                    ; two-second timeout

> (print-move "Best␣move"         ; print best move
    best-move)

> (define state                   ; apply best move to state
    (move-state best-move))

> (define state                   ; apply the fourth valid move
    (move-state                   ; to state
      (list-ref
        (valid-moves state)
        3)))
```

3 Notes on the TUI

- The board comprises an 8×8 black-and-white board. The checkers can only fall on the black squares. Black (player 1) starts on the top and its men move downward; white (player 2) starts on the bottom and its men move upward.
- Black (player 1) pieces are represented by non-solid circles and kings, and white (player 2) pieces are represented by solid circles and kings.
- The possible moves are listed and numbered. The coordinates are in (row, column) order; both row and column are zero-indexed.
- There is not very strict user input validation and error checking, but most inputs should behave as expected (e.g., throw an error if the input state file is not found). When prompting for a move, the program will keep re-prompting until a valid move number is entered.
- Ctrl+C should be sufficient to exit the game at any point. (If running in the Scheme REPL, you may need to press Ctrl+C multiple times, and this will enter an interrupt handler; exit this by entering "q".)

4 Main algorithm implementation details

Representation of states and moves Since there are 32 playable squares on a checkers board, we can represent the positions of the black checkers and of the white checkers (regardless of

their type) as two 32-bit bitmasks. The player's pieces are also stored in two arrays of length 12 (maximum of twelve checkers per player), where each element includes the checker type and coordinates (these values are packed into one byte). The former representation allows for constant-time lookup of a board position, and the latter allows for efficient iteration over the players' checkers. Each state object contains these two fields (boards, 8 bytes; pieces, 24 bytes) as well as which player has the next move.

Generating new moves The valid-moves function first checks for capture moves, and if there are none, then checks for non-capture moves. Capture moves are implemented by using DFS with a stack of intermediate moves.

Minimax search with alpha-beta pruning This was taken almost directly from the textbook. The min-value function only returns the best value, while the max-value returns both the best value and the best move; otherwise they are the same. (I didn't implement negamax.)

Time-based iterative deepening At the top of the max-value and min-value functions, there is a check for whether the time limit will be reached within 10 milliseconds. In the C++ version, a special value is passed up the stack; in the Scheme version, a continuation directly passes control back to the iterative-deepening driver function.

Heuristic function The score is calculated as follows (in order of importance):

king and men count number of kings and men, kings are worth more than men

men distance to king closeness of men to being promoted to kings

back row how many men are in the back row; higher is generally better as it doesn't allow enemies to get promoted as easily

trade affinity if winning, try to maximize the ratio of the number of checkers remaining

sum of distances to corners only used in endgame position, used to chase pieces out of double corners

board center how close the pieces are to the center, where there is more freedom and less of a chance to get trapped

randomization used to easily break ties and make the game nondeterministic

This is a simple heuristic that does fairly well, but it may not see far enough in certain endgame positions to make a decisive win when it might be clear to the human eye. For example, in some cases where a player has a three-to-two piece advantage, it may not efficiently dispatch the pieces to capture the losing side's pieces one by one.

Early stopping There are two conditions when the iterative deepening algorithm stops before the time limit is reached. Namely, this is if there is only one valid move, or if the entire game tree is searched (i.e., a definite win or definite loss is determined). From a practical view, it may be better to stall even if a definite loss is determined so that the match lasts longer, but this implementation does not prolong the inevitable.

5 Extra features

- The way I implemented the game, it was very easy to implement an undo function. In short, a list of the past game states is stored as the game progresses, and it is easy to revert to a prior state by setting the current state to the stored past state. This was implemented in both versions of the game.
- In the Scheme version, I also added the option to let iterative deepening choose the next move instead of manually choosing a move, as well as an option to quit the game.

6 General comments

- The C++ version was written first, and the logic was translated directly into the Lisp version. It took roughly two days to plan how to generate the moves (and the relevant data structures), two days to implement the main algorithm in C++, three days to research and design a simple but satisfactory heuristic, and another three days to rewrite the main program in Scheme. Then I spent about a week on general touch-ups.
- I realized I could make some additional optimizations when doing the Scheme version (e.g., a 32-bit bitmask suffices to store the board state, 64-bits is not necessary; and only one bitmask is needed for each player, I used two in the C++ implementation).
- In the C++ version, I was worried about saving space at first (originally I had the idea that we were to save the entire tree in memory, which I realized was false after starting to implement the project), so there are a lot of 8-bit integral data types thrown around. In Lisp, everything is of the fixnum type (60-bit integral type).
- I tried to avoid dynamic memory allocation completely in C++; everything is allocated on the stack (i.e., in small fixed-size buffers). The buffer lengths were somewhat arbitrarily chosen so that they weren't too large but were large enough for any reasonable gameplay.
- In Scheme, there is a lot less control over memory allocation, so I used the defaults for memory allocation. I tried to stick to immutable values to follow the functional paradigm of Lisp, but I broke this when modifying the bytevector buffers (board bitmasks) in place when necessary. I also used linked lists for storing the list of available moves rather than an array buffer. I'm not sure how costly Lisp's memory allocation is; I believe it uses slab allocation to improve performance, but it is likely a major reason that the Scheme version is slower than the C++ version.
- In general, the C++ code felt a lot more terse and easy to write. However, one thing that was really nice about the Lisp implementation was the use of (first-class) continuations; this allows for a really clean and easy way to return control to the iterative deepening function from anywhere in the minimax search when time runs out. In the C++ version, you either have to traverse all the way up the call stack or do some unfavorable signaling or cross-function jumps (e.g., setjump/longjump).
- The colors were set using ANSI escape sequences, and the printed board pieces are unicode. Make sure to use a terminal emulator that supports these ANSI escapes (most *nix terminals do) and uses a monospace font.

- From my very-informal tests, the C++ version usually gets to the same depth or outperforms the Scheme version by fewer than three extra depths. I also loosely benchmarked the valid-moves function on some sample states, and found that the C++ version generates nodes roughly four times as quickly, which could correspond to no or one extra search depths.
- On all of the toy states (the provided states 1 through 10), both versions play well with only one second timeouts. In the case of an ordinary game of the AI playing against itself, it is not uncommon to see wins or losses on either side, nor is it uncommon to see draws.

References

[1] http://www.chesslab.com/rules/CheckerComments3.html