# ComS 327 Project Part 3

Checkers: rank available moves

Fall 2019

## 1 Summary

For the third part of the project, you must read an input file that describes a game configuration, determine all moves possible for the current player (i.e., the player whose turn it is currently), and determine a score for each of these moves. The score is based on how well the current player can expect to do looking $D$ moves ahead, assuming the opponent plays perfectly, where $D$ is a user-specified number of moves.

Specifically, your `Makefile` must now build the executables for parts 1 and 2, and a new executable named `rankmoves`. Your `rankmoves` program will read an input file in exchange format, and write the list of moves to standard output in the format specified below. Program `rankmoves` should accept the following optional command-line switches and arguments, which may appear in any order.

- The input filename, guaranteed not to begin with a `'-'` character. If omitted, read from standard input.
- Switch `-d D`, indicating that at most $D$ lookahead moves (or "depth") should be considered. If omitted, $D$ should be 0.
- Switch `-v`, indicating *verbose* output.

You may implement additional switches to help test or debug your code, or to provide extra or different features, if you wish.

Your code will be tested only on exchange files with no moves given. If an input file specifies moves, you may either:

- Exit cleanly with an appropriate error message, to standard error.
- Proceed, ignoring the specified moves (with an appropriate warning to standard error).
- Process the moves, and then proceed on the resulting game configuration.

## 2 Move scoring

For this project, the following simple scoring functions are used to determine a score for a game configuration $G$:

$$s_{red}(G) = \begin{cases} -99 & \text{if red has no moves} \\ 99 & \text{if black has no moves} \\ (\#\text{r} + 2 \cdot \#\text{R}) - (\#\text{b} + 2 \cdot \#\text{B}) & \text{otherwise} \end{cases}$$

$$s_{black}(G) = \begin{cases} -99 & \text{if black has no moves} \\ 99 & \text{if red has no moves} \\ (\#\text{b} + 2 \cdot \#\text{B}) - (\#\text{r} + 2 \cdot \#\text{R}) & \text{otherwise} \end{cases}$$

In the formulas, #r refers to the number of red pawns on the board, #R refers to the number of red kings on the board, #b refers to the number of black pawns on the board, and #B refers to the number of black kings on the board. Note that for any given game configuration, the score for red is the negative of the score for black, and a score of 99 means that it is a winning configuration.

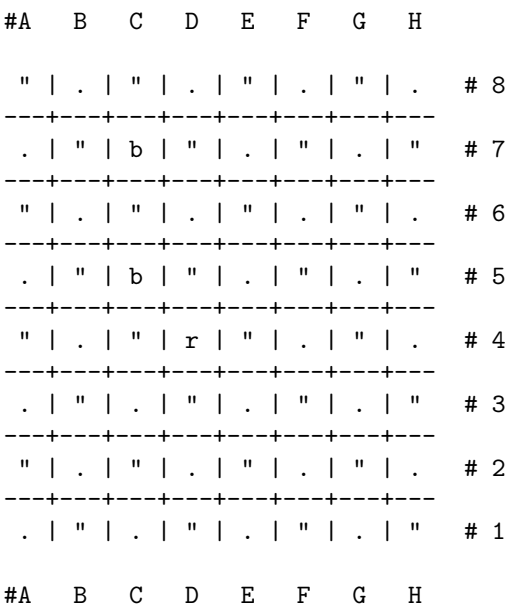


Figure 1: Initial board configuration for example discussions.

For the discussion below, I write

$$G \stackrel{m}{\rightarrow} G'$$

to indicate that, from a game configuration $G$, if move $m$ is played, then the new game configuration will be $G'$.

## 2.1 No lookahead ($D = 0$)

If the current game configuration is $G$ and the lookahead depth $D$ is zero, the moves for the current player $t$ are scored as follows. For each possible move $m$ that the current player could make in configuration $G$, the score for move $m$ is $s_t(G')$, where $G \stackrel{m}{\rightarrow} G'$. In words, from the current configuration, determine the new configuration after move $m$ is taken, and score that configuration using the scoring function of current player (i.e., the one taking the move).

As an example, consider the board configuration shown in Figure 1. Then, if it is red's turn, there are two moves available: `d4->b6` and `d4->e5`. Using $D = 0$, these moves are scored as follows. For `d4->b6`, after taking the move, there is one red pawn and one black pawn, giving a score of 0. For `d4->e5`, after taking the move, there is one red pawn and two black pawns, giving a score of -1.

## 2.2 With lookahead

Suppose now the lookahead depth $D$ is more than zero. A move $m$ for the current player is scored as follows. If the current game configuration is $G$, determine $G'$ where $G \stackrel{m}{\rightarrow} G'$. Then, for configuration $G'$, determine all moves available for the opponent player, and score them with a lookahead of $D-1$. If there are no moves available for the opponent, then move $m$ has a score of 99. Otherwise, assume that the opponent chooses a move that will maximize her score (or, equivalently, minimize the current player's score). The score for move $m$ is then the best outcome score the current player can expect from configuration $G'$.

As an example, let us examine red's moves with $D = 1$ for the game configuration below (which is the same as Figure 1).

```
" . " . " . " .
. " b " . " . "
" . " . " . " .
. " b " . " . "
" . " r " . " .
. " . " . " . "
" . " . " . " .
. " . " . " . "
```

Again, the possible moves are `d4->b6` and `d4->e5`. Taking `d4->b6`, we reach configuration

```
" . " . " . " .
. " b " . " . "
" r " . " . " .
. " . " . " . "
" . " . " . " .
. " . " . " . "
" . " . " . " .
. " . " . " . "
```

which is now explored using $D = 0$ for black. Black has two moves available: `c7->a5`, with a score of 99 for black, and `c7->d6`, with a score of 0 for black. Since we assume black will play perfectly, black will choose move `c7->a5` and score 99. Thus, the score for red's move `d4->b6` is -99, since this is the expected outcome of taking that move.

If red instead takes move `d4->e5`, we reach configuration

```
" . " . " . " .
. " b " . " . "
" . " . " . " .
. " b " r " . "
" . " . " . " .
. " . " . " . "
" . " . " . " .
. " . " . " . "
```

which again is explored using $D = 0$ for black. Black has four moves available: `c5->b4`, `c5->d4`, `c7->b6`, and `c7->d6`. Taking each of these moves will give a different configuration, each with a score of 1 for black. Thus, the score for red's move `d4->e5` is -1.

# 3 Output format

If there are no moves available, output should be

`Player (player) has no moves available.`

Otherwise, all current moves for the current player should be displayed, **in alphabetical order** (so that it is easier to compare outputs). If the verbose flag is set, all considered moves at each depth should be displayed, **in alphabetical order**. The detailed output formats are given below.

## 3.1 Non-verbose output

Output should be a list of moves, with scores, for the current player. The output format should be

`(move) for (player): score (value)`

with each move on its own line.

### 3.2 Verbose output

If the current depth is greater than zero, then display the following when starting to consider a move:

`(indent)? (move) for (player):`

where (indent) indicates a number of spaces for indenting purposes (discussed below). Then, from the configuration reached after that move, the list of moves for the other player should be shown, in the same format but with increased indentation. After the score is determined for the move, display:

`(indent). (move) for (player): score (value)`

where the level of indentation should match the corresponding "`?`" output line for the move, if any. The indentation should initially be zero spaces, and should increase by 4 spaces as moves are processed.

## 4 Example outputs

### 4.1 Input file: `rankin.txt`

We use an input file consistent with the example shown in Figure 1 and the earlier discussions:

```
RULES:
no capture
single jumps
TURN:
red
BOARD:
 " | . | " | . | " | . | " | .   # 8
---+---+---+---+---+---+---+---
 . | " | b | " | . | " | . | "   # 7
---+---+---+---+---+---+---+---
 " | . | " | . | " | . | " | .   # 6
---+---+---+---+---+---+---+---
 . | " | b | " | . | " | . | "   # 5
---+---+---+---+---+---+---+---
 " | . | " | r | " | . | " | .   # 4
---+---+---+---+---+---+---+---
 . | " | . | " | . | " | . | "   # 3
---+---+---+---+---+---+---+---
 " | . | " | . | " | . | " | .   # 2
---+---+---+---+---+---+---+---
 . | " | . | " | . | " | . | "   # 1
MOVES:
```

### 4.2 Example run 1

If we run

```
./rankmoves -v < rankin.txt
```

we should get the following on standard output:

```
. d4->b6 for red: score 0
. d4->e5 for red: score -1
```

### 4.3 Example run 2

If we run

```
./rankmoves -v rankin.txt -d 1
```

we should get the following on standard output:

```
? d4->b6 for red:
    . c7->a5 for black: score 99
    . c7->d6 for black: score 0
. d4->b6 for red: score -99
? d4->e5 for red:
    . c5->b4 for black: score 1
    . c5->d4 for black: score 1
    . c7->b6 for black: score 1
    . c7->d6 for black: score 1
. d4->e5 for red: score -1
```

### 4.4 Example run 3

If we run

```
./rankmoves -v rankin.txt -d 2
```

we should get the following on standard output:

```
? d4->b6 for red:
    ? c7->a5 for black:
    . c7->a5 for black: score 99
    ? c7->d6 for black:
        . b6->a7 for red: score 0
        . b6->c7 for red: score 0
    . c7->d6 for black: score 0
. d4->b6 for red: score -99
? d4->e5 for red:
    ? c5->b4 for black:
        . e5->d6 for red: score -1
        . e5->f6 for red: score -1
    . c5->b4 for black: score 1
    ? c5->d4 for black:
        . e5->d6 for red: score -1
        . e5->f6 for red: score -1
    . c5->d4 for black: score 1
    ? c7->b6 for black:
        . e5->d6 for red: score -1
        . e5->f6 for red: score -1
    . c7->b6 for black: score 1
    ? c7->d6 for black:
        . e5->c7 for red: score 0
        . e5->f6 for red: score -1
    . c7->d6 for black: score 0
. d4->e5 for red: score -1
```

### 4.5 Example run 4

If we run

```
./rankmoves rankin.txt -d 2
```

we should get the following on standard output:

```
d4->b6 for red: score -99
d4->e5 for red: score -1
```

In general, when the `-v` flag is not set, the output should be the "." lines with zero spaces of indentation of the verbose output, formatted slightly differently.

## 5 Required functionality

Your `rankmoves` program should work correctly for "standard boards", "single jumps", and "no capture" (i.e., players are not required to capture a piece). If an input file specifies a feature that you have not implemented, your code should exit cleanly with an appropriate error message to standard error (something like, "Sorry, multiple jumps are not implemented").

If you are working in a group of 2, you must complete at least one extra-credit feature (see below).

## 6 Extra credit features

- Allow input files that specify the "capture" rule, and adjust game play accordingly (i.e., in a board configuration where a capture is possible, non-capture moves should not be considered).
- Allow input files that specify "multiple jumps", and adjust game play accordingly (i.e., you must consider multiple jump moves, if such a move is possible).

Note that handling "flipped boards" is *not* worth extra points for this part of the project.

## 7 What to submit

Remember to submit the following in your git repository.

- Source code, in C and/or C++.
- A `Makefile`, so your executables can be built by simply typing "`make`".
- A `README` file, to indicate which features are implemented.
- A `DEVELOPERS` file, with necessary information about the source code for other developers. For group submissions, this file also should indicate which student(s) implemented which function(s).
- A `GRADEME` file, containing either

  > "Please grade my updated part 2 submission, for at most half credit. I understand that this could lower my grade for part 2."

  OR

  > " "

  An omitted file will be treated the same as an empty file.

Also, please turn in a short note in Canvas with the URL of your git repository and the members of your group.