CSCI-561 – Spring 2021 - Foundations of Artificial Intelligence
Homework 2

Due March 17, 2021 23:59:59

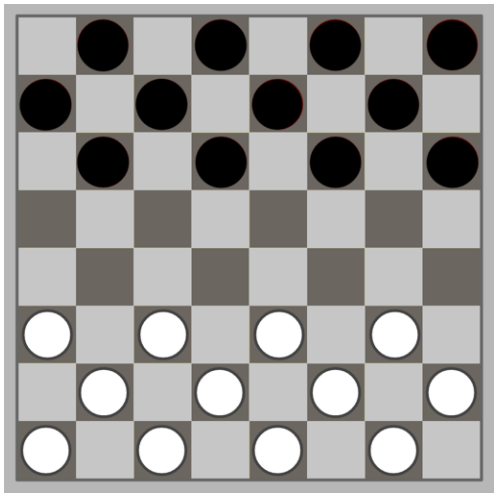


Image from controlaltachieve.com

Guidelines

This is a programming assignment. You will be provided sample inputs and outputs (see below). Please understand that the goal of the samples is only to check that you can correctly parse the problem definitions and generate a correctly formatted output that contains a valid, but not necessarily good (or the only possible) move. In most situations, several moves will be possible, so your move may differ from our example and still be perfectly valid. You should not assume that if your program works on the samples, it will work on all test cases. It is your task to make sure that your program will work correctly on any valid input. You are encouraged to try your own test cases or have your program play against itself to check how your program would behave in some complex situations. Since **each homework submission is checked by a set of programs**, your output should match the specified format ***exactly***. Failure to do so will most certainly cost some points. The output format is simple and examples are provided. You should upload and test your code on vocareum.com, and you will submit it there. You may use any of the programming languages and versions thereof provided by vocareum.com.

Grading

Your code will be tested as follows: Your program should not require any command-line argument. It should read a text file called “input.txt” in the current directory that contains a problem definition. It should write a file “output.txt” with your solution to the same current directory. Format for input.txt and output.txt is specified below. End-of-line character is LF (since vocareum is a Unix system and follows the Unix convention).

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, **you will get zero points**. Anything you write to stdout or stderr will be ignored and is ok to leave in the code you submit (but it will likely slow you down). Please test your program with the provided sample files to avoid any problem.

Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe you are urged to err on the side of caution. Do not copy work from another student or off the web. Keep in mind that sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

> **Do not copy** code or written material from another student. Even single lines of code should not be copied.
> **Do not collaborate** on this assignment. The assignment is to be solved individually.
> **Do not copy** code off the web. This is easier to detect than you may think.
> **Do not share** any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.
> **Do not copy** code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homework submissions of previous years.
> **Do not ask on piazza** how to implement some function for this homework, or how to calculate something needed for this homework.
> **Do not post code on piazza** asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
> **Do not post test cases on piazza** asking for what the correct solution should be.
>
> **Do** ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

Project description

In this project, we will play the game of ***Checkers***, the classic strategy board game. It is a version of the game *draughts* and is also called **American checkers** or **straight checkers**. It uses an 8x8 checkered gameboard. Each player starts with 12 game pieces placed on the dark squares of the side of the gameboard closest to them, as can be seen in the figure above. The side with the darker colored pieces is usually called 'Black' and the side with the lighter color is 'White'. Black opens the game. The pieces move diagonally forward and can capture opponent's pieces by jumping over them. Whenever a piece reaches the opposite side of the board, it is crowned ***king*** and gains the ability to move/capture both forward and backward. The game ends when one side wins or a draw condition applies.

One side wins when the opponent cannot make a move, which can happen in two ways: 1) All the opponent's pieces have been captured. 2) All the opponent's pieces are blocked in some way. A draw also applies in two cases: 1) Over 50 consecutive turns, there has been no material change on the board (no pieces were captured or crowned), or 2) The same exact position on the board has been reached 3 times (i.e., all pieces of the two players combined have been in the exact same places 3 times).

More details on the game can be found at https://en.wikipedia.org/wiki/English_draughts and we will also go over the gameplay for you below. Beware that many variants of this game exist, so please adhere to the rules described in this homework carefully.

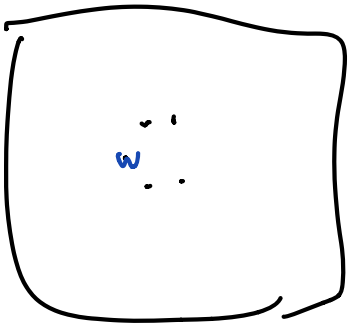
This 8x8 variant of checkers is *solved*. Which means that when both players play a perfect game, the game comes to a draw. If you're interested, you can check out the paper at the following link: https://www.researchgate.net/publication/231216842_Checkers_Is_Solved
This also means that checkers doesn't have a first or second player advantage, and it will always lead to a draw with perfect play. Note that we do not expect your program to play perfectly, nor will this be necessary to succeed at this assignment, but we wanted to note this so that you know that there is no advantage to the player that gets the first move.

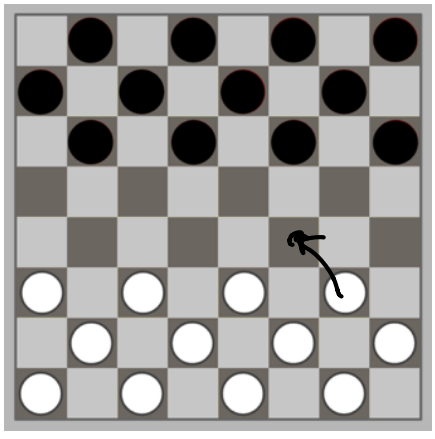
Setup of the game:

The setup of the game is as follows:

- Simple wooden pawn-style playing pieces, sometimes called '*men*'.
- The board consists of an 8x8 grid of squares.
- Players' sides are situated across the board from each other.
- The first row of squares on each player's side is usually called the '*king's row*'. When an opponent's piece reaches this row, it will be crowned '*king*'.
- Each player has a set of 12 pieces in a distinct color, which are placed on the 12 closest dark squares on their side of the board.

Here's the visualization of this setup, as we will use in this homework:





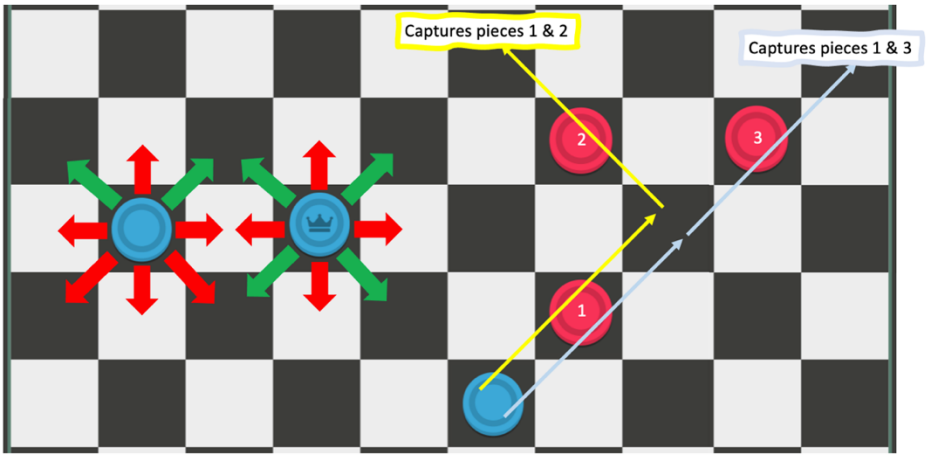
Play sequence:

We first describe the typical play for humans. We will then describe some minor modifications for how we will play this game with artificial agents.

- Create the initial board setup according to the above description.
- Players randomly determine who will play black/or the darker color. Black will play first.
- Pieces can move only diagonally. Regular pieces can only move forward. Pieces that have previously been crowned 'king' by reaching the king's row of their opponent can move diagonally both forward and backward.
- Each player's turn consists of moving a single piece of one's own color in one of the following plays:
    - A simple move:
        - Move the piece to any diagonally forward adjacent empty square. (King pieces can also move to one of the adjacent backward diagonals.) **(Note: In this kind of play, the pieces can move only one square.)**
        - This move ends this player's turn, even if the move results in a position that makes one or more subsequent jumps possible (see below for jumps).
    - One or more jumps, capturing one or more of the opponent's pieces **(Note: If a jump is possible at any point in the turn, it is mandatory)**:
        - An adjacent piece of the opponent in any of the allowed diagonal directions, i.e., forward-left and forward-right (and also backward-left and backward-right for king pieces) can be jumped over if there is an empty square on the other side of that piece.
        - After the jump, the piece that was jumped over is "captured" (eliminated from the game).
        - Place the piece in the empty square on the opposite side of the jumped piece.
        - All jumping moves are compulsory. Every opportunity to jump must be taken. In the case where there are different jump sequences available, the player may choose which sequence to make, whether it results in the most pieces being taken or not. (This means that a player is not allowed to make a sequence of one or more jumps whose end result would still allow her/him/it to jump some more.)
- If a piece has reached the opposing king's row, it is crowned 'king' and can now also move backwards in following turns.
- If the current play results in a board where the opponent has no pieces left or cannot move any of their remaining pieces, the acting player wins. Otherwise, play proceeds to the other player.

In the image below, we show examples of valid moves (in green) and invalid moves (in red) for a regular and a king piece, as well as the different jump scenarios for a given toy scenario. A regular piece can only move in forward diagonals while a 'king' (with the crown symbol on it) can also move backward.

For the simple jump scenario given on the right, the blue piece has to choose between two different jump scenarios. After jumping over red piece 1, it can either continue jumping right and capture red piece 3 or choose to go left instead and capture red piece 2. Therefore, it is possible for the blue piece to switch directions as it fulfils a jump sequence during a play. Note that, the blue piece cannot stop after making the jump over red piece 1 since, if a jump is possible at any point in the play, it has to be made.



Playing with agents

In this homework, your agent will play against another agent, either implemented by the TAs, or by another student in the class. For grading, we will use two scenarios:

1) **Single move:** Your agent will be given in input.txt a board configuration, a color to play, and some number of seconds of allowed time to play one move. Your agent should return in output.txt the chosen move(s), before the given play time has expired. Play time is measured as total CPU time used by your agent on all CPU threads it may spawn (so, parallelizing your agent will not get you any free time). Your agent will play 10 single moves, each worth one point. If your agent returns an illegal move, a badly formatted output.txt, or does not return before its time is up, it will lose the point for that move.
2) **Play against reference agent:** Your agent will then play 9 full games against a simple minimax agent with no alpha-beta pruning, implemented by the TAs. There will be a limited total amount of play time available to your agent for the whole game (e.g., 100 seconds), so you should think about how to best use it throughout the game. This total amount of time will vary from game to game. Your agent must play correctly (no illegal moves, etc.) and beat the reference minimax agent to receive 10 points per game. Your agent will be given the first move on 5 of the 9 games. In case of a draw, the agent with more remaining play time wins.
   Draw condition: The game will be called a draw if for 50 consecutive turns: 1) The number of pieces on the board haven't changed, and 2) The status of the pieces on the board

haven't changed (i.e., none has been crowned king). A draw will also be called if the same exact position on the board has been reached 3 times.

Note that we make a difference between single moves and playing full games because in single moves it is advisable to use all remaining play time for that move. While playing games, however, you should think about how to divide your remaining play time across possibly many moves throughout the game.

In addition to grading, we will run a competition where your agent plays against agents created by the other students in the class. This will not affect your grade. The top agents will be referred to a contact at Google for an informal introduction. There will also be a prize for the grand winner.

Agent vs agent games:

Playing against another agent will be organized as follows (both when your agent plays against the reference minimax agent, or against another student's agent):

A master game playing agent will be implemented by the grading team. This agent will:

- Create the initial board setup according to the above description.
- Assign a player color (black or white) to your agent. The player who gets assigned black will have the first move.
- Then, in sequence, until the game is over:
    - The master game playing agent will create an input.txt file which lets your agent know the current board configuration, which color your agent should play, and how much total play time your agent has left. More details on the exact format of input.txt are given below.
    - We will then run your agent. Your agent should read input.txt in the current directory, decide on a move, and create an output.txt file that describes the move (details below). Your time will be measured (total CPU time). If your agent does not return before your time is over, it will be killed and it loses the game.
    - Your remaining playing time will be updated by subtracting the time taken by your agent on this move. If time left reaches zero or negative, your agent loses the game.
    - The validity of your move will be checked. If the format of output.txt is incorrect or your move is invalid according to the rules of the game, your agent loses the game.
    - Your move will be executed by the master game playing agent. This will update the game board to a new configuration.
    - The master game playing agent will check for a game-over condition. If one occurs, the winning agent or a draw will be declared accordingly.
    - The master game playing agent will then present the updated board to the opposing agent and let that agent make one move (with the same rules as just described for your agent; the only difference is that the opponent plays the other color and has its own time counter).

Input and output file formats:

Input: The file input.txt in the current directory of your program will be formatted as follows:

First line: A string SINGLE or GAME to let you know whether you are playing a single move (and can use all of the available time for it) of playing a full game with potentially many moves (in which case you should strategically decide how to best allocate your time across moves).

Second line: A string BLACK or WHITE indicating which color you play. Black will always start the play with its pieces placed at the top of the game board, with white on the bottom, as given in the above images.

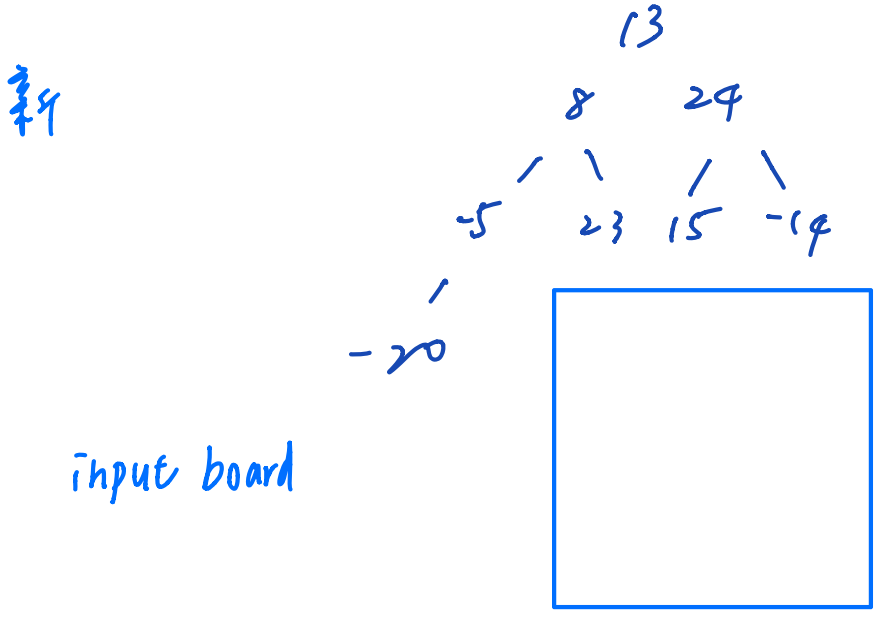
Third line: A strictly positive floating point number indicating the amount of play time remaining for your agent (in seconds).

Next 16 lines: Description of the game board, with 16 lines of 16 symbols each:

- `w` for a grid cell occupied by a white regular piece
- `W` for a grid cell occupied by a white king piece
- `b` for a grid cell occupied by a black regular piece
- `B` for a grid cell occupied by a black king piece
- `.` (a dot) for an empty grid cell

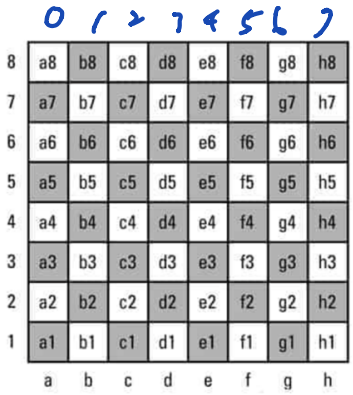
For example:

```
SINGLE
WHITE
100.0
.b.b.b.b
b.b.b.b.
.b...b.b
....b...
........
w.w.w.w.
.w.w.w.w
w.w.w.w.
```



In this example, your agent plays a single move as white color and has 100.0 seconds. The board configuration is just the one from the start of the game, after black has made the first move.

Output: The format we will use for describing the square positions is borrowed from the algebraic notation used for chess, where every column is described by a letter and every row is described by a number. The position for a given square is given as the concatenation of these. Here's a useful visualization on how we identify each square for the 8x8 checkers board:



8	a8	b8	c8	d8	e8	f8	g8	h8
7	a7	b7	c7	d7	e7	f7	g7	h7
6	a6	b6	c6	d6	e6	f6	g6	h6
5	a5	b5	c5	d5	e5	f5	g5	h5
4	a4	b4	c4	d4	e4	f4	g4	h4
3	a3	b3	c3	d3	e3	f3	g3	h3
2	a2	b2	c2	d2	e2	f2	g2	h2
1	a1	b1	c1	d1	e1	f1	g1	h1
	a	b	c	d	e	f	g	h

Image from ichess.net

As an example, in the input sample given above, black has moved their piece from position d6 to e5. Note that for checkers, valid moves will always land in dark squares.

Using the above notation for the squares on our gameboard, the file output.txt which your program creates in the current directory should be formatted as follows:

1 or more lines: Describing your move(s). There are two possible types of moves (see above):

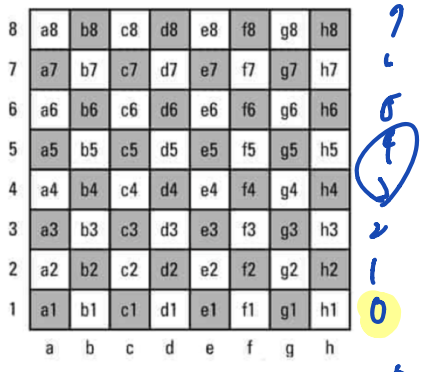
> **E FROM_POS TO_POS** – Your agent moves one of your pieces from location FROM_POS to an adjacent empty location (on the diagonal) TO_POS. FROM_POS and TO_POS will be represented using the notations explained above, by a lowercase letter from a to h and a number from 1 to 8. As explained above, TO_POS should be adjacent to FROM_POS (on the diagonal) and should be empty. If you make such a move, you can only make one per turn.
>
> **J FROM_POS TO_POS** – Your agent moves one of your pieces from location FROM_POS to empty location TO_POS by jumping over a piece in between. You should write out **one jump per line** in output.txt if your play results in more than one jumps.

For example, output.txt may contain:

```
E c3 b4
```

The resulting board would look like this, given the above input.txt:

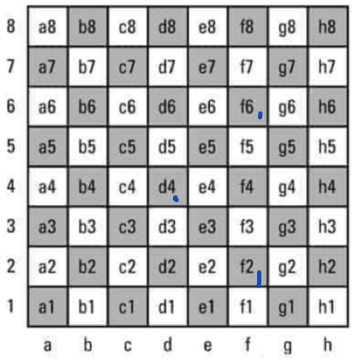
```
.b.b.b.b
b.b.b.b.
.b...b.b
....b...
.w......
w.....w.
.w.w.w.w
w.w.w.w.
```



8	a8	b8	c8	d8	e8	f8	g8	h8
7	a7	b7	c7	d7	e7	f7	g7	h7
6	a6	b6	c6	d6	e6	f6	g6	h6
5	a5	b5	c5	d5	e5	f5	g5	h5
4	a4	b4	c4	d4	e4	f4	g4	h4
3	a3	b3	c3	d3	e3	f3	g3	h3
2	a2	b2	c2	d2	e2	f2	g2	h2
1	a1	b1	c1	d1	e1	f1	g1	h1
	a	b	c	d	e	f	g	h

Let's look at another example that consists of a jump sequence. Let's say input.txt is as given below:

```
SINGLE
BLACK
100.0
.b.....b
b...b.b.
.b...b.b
..b.w...
........
w...w...
.w.w...w
w...w.w.
```



blacks

whites

2-6 = -4

-4+2 = 2

fmp_board = board

The file output.txt will contain the following, since all possible jumps have to be taken:

```
J f6 d4
J d4 f2
```

each piece → map<pos, skip>

↳ change board and push to vector

⇓

by changing only piece and move

After which the board would look like:

```
.b.....b
b...b.b.
.b.....b
..b.....
........
w.......
.w.w.b.w
w...w.w.
```

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose ("py" for python, "cpp" for C++, and "java" for Java). If you are using C++11, then the name of your file should be "homework11.cpp" and if you are using python3 then the name of your file should be "homework.py". If you are using python2, even though it is now officially unsupported, you can name your file "homework2.py".
- The board you will be given as input will always be valid and will have twelve or fewer `w`/`W` letters, twelve or fewer `b`/`B` letters, and the rest will be `.` (standing for empty cells).
- Likely (but not guaranteed), total play time will be 5 minutes (300.0 seconds) when playing against another agent, and 30.0 seconds for single moves.

- Play time used on each move is the total combined CPU time as measured by the Unix **`time`** command. This command measures pure computation time used by your agent, and discards time taken by the operating system, disk I/O, program loading, etc. Beware that it cumulates time spent in any threads spawned by your agent (so if you run 4 threads and use 400% CPU for 10 seconds, this will count as using 40 seconds of allocated time).
- If your agent runs for more than its given play time (in input.txt), it will be killed and will lose the single move or the game.
- You need to think and strategize how to best use your allocated time. In particular, you need to decide on how deep to carry your search, on each move. In some cases, your agent might be given only a very short amount of time (e.g., 5.2 seconds, or even 0.01 seconds), for example towards the end of a game. Your agent should be prepared for that and return a quick decision to avoid losing by running over time. The amount of play time that will be given in input.txt will always be >0, but it could be very small if you are close to running out of time.
- To help you with figuring out the speed of the computer that your agent runs on, you are allowed to also provide a second program called **calibrate.xxx** (same extension conventions as for homework.xxx). This is optional. If one is present, **we will run your calibrate program once (and only once)** before we run your agent for grading or against another agent. You can use calibrate to, e.g., measure how long it takes to expand some fixed number of search nodes, or to benchmark file I/O. You can then save this into a single file called **calibration.txt** in the current directory. When your agent runs during grading or during a game, it could then read calibration.txt in addition to reading input.txt, and use the data from calibration.txt to strategize about search depth or other factors.
- You need to think hard about how to design **your eval function** (which gives a value to a board when it is not game over yet).
- You are allowed to maintain persistent data across moves during a game, by writing such data to a single file called **playdata.txt** in the current directory. Before a new game starts, the master game playing agent will delete any playdata.txt file. So, on your first move, this file will not exist, and you should be prepared for that. Then, you can write some data to that file at the end of a move and read that file back at the beginning of the next move.
- There is no first or second player advantage in this game for two agents with perfect play. In our case, when playing against the reference minimax agent we will give your agent the first move for 5 of the 9 games. In the competition, we will play two agents against each other for an even number of games, and advance both agents to the next round of the competition if they both win or draw on half of the games. If an agent loses more than half of the games, it will be eliminated and only the other agent will move to the next round of the competition. We may end up with several equivalent winners of the competition.

Example 1:

For this input.txt:

```
SINGLE
BLACK
100.0
.b...b..
..b...b.
.....w.b
....w...
.b......
w...w.w.
...B....
......w.
```

One possible correct output.txt is:

```
J d2 f4
J f4 d6
```

Here, black can only move backwards because the piece in d2 has been crowned king in a previous turn.

Example 2:

```
SINGLE
WHITE
6.6
........
w...b.b.
........
..w.b...
.w...b..
....b...
.w.....b
..w...w.
```

One possible correct output.txt is:

```
E a7 b8
```

Which results in a white piece reaching the king's row of the opponent and getting crowned king.

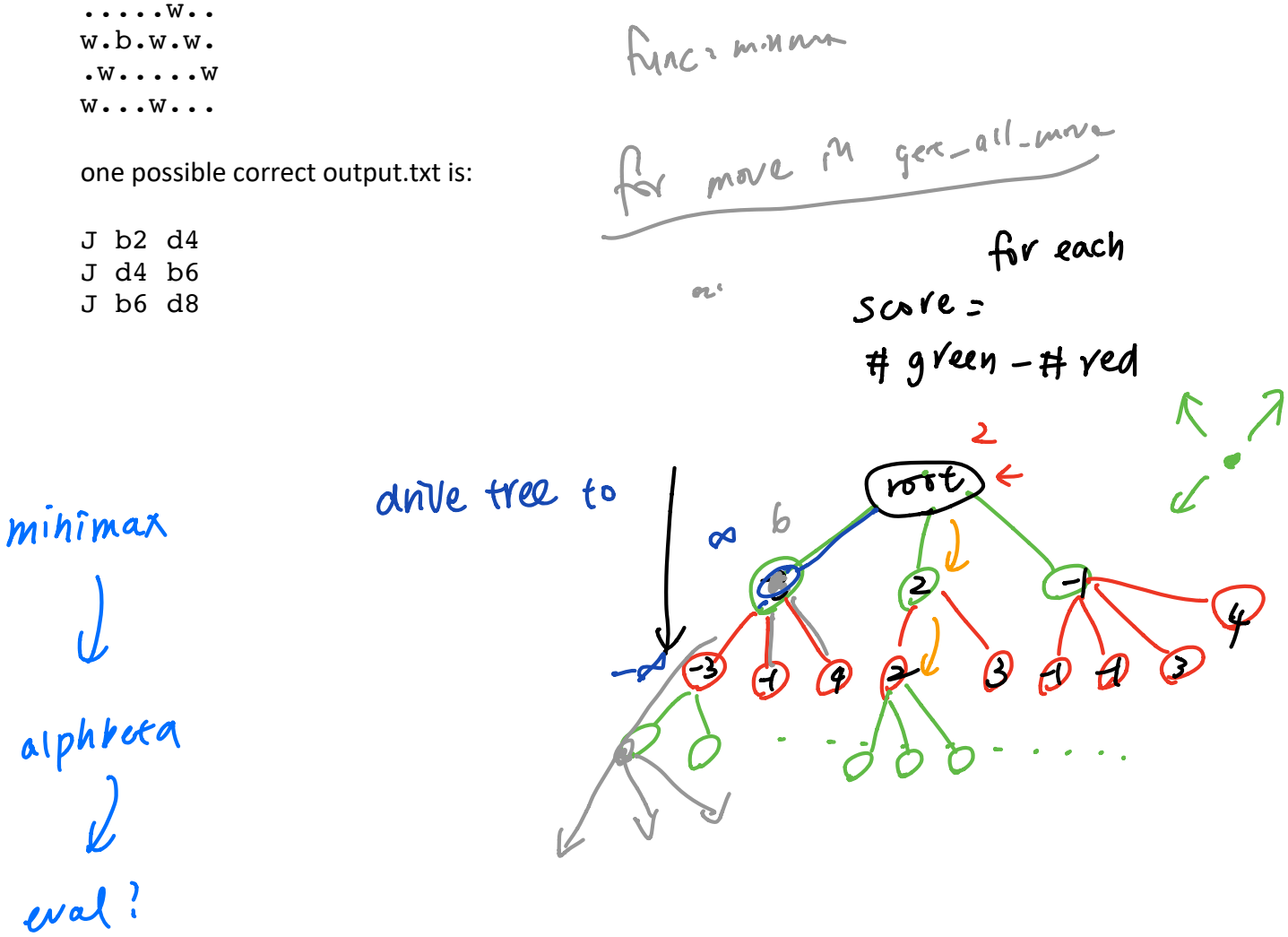
Example 3:

For this input.txt:

```
SINGLE
WHITE
23.33
.b...b..
..b.b.b.
...b...b
..b.....
.....w..
w.b.w.w.
.w.....w
w...w...
```

one possible correct output.txt is:

```
J b2 d4
J d4 b6
J b6 d8
```



hold piece info.

board - check valid

white_left

black_left

white_king

black_kings

game

update
winner
reset
select
draw_valid_move

piece class.

board class

init_board

- move: move piece to row_colu
- get_piece (row, col)
- create_board( )

- draw
- remove
- get_valid_moves
- traverse_left
- traverse_right

↓

eval method ✡ better eval, better AI

(4) white_left - red_left
+(white_king * 0.5 - red_king * 0.5)

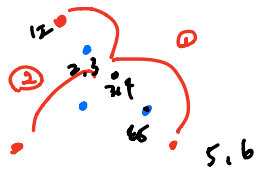
get_all_pieces (color)

valid move

moves = {{}} piece:

① if diagnal empty → could move

else if



map <pos, vector (key, vector)> last double ↑ jump

key →

① last(3,4,white)

②

input board

↓

b - piece

w - piece

try to move all