CSC111 Assignment 2: Trees, Chess, and Artificial Intelligence

Over the past few weeks, you've learned about the programming technique of *recursion* and various recursive data types, such as nested lists, `RecursiveList`, and trees. On this assignment, you'll apply what you've learned to one fundamental application of trees: representing the state of a system, and branching decisions that lead from one state to another. The domain that we've chosen for this assignment is *chess*, for three main reasons:

1. It is a "perfect information" game, meaning that both players always have access to the full game state, without any hidden information or randomness.
2. There are some robust Python chess libraries that we can take advantage of to simplify our code while still making a fun and interactive program.
3. Chess is one of the most commonly-studied games in artificial intelligence, with a rich history of [chess engines (https://en.wikipedia.org/wiki/Computer_chess)](https://en.wikipedia.org/wiki/Computer_chess) that have consistently beaten the top human chess players for over two decades.

Now, this assignment will focus on a simplified version of chess known as *Minichess*, to make the computations and algorithms accessible for you on your computer. However, the techniques we'll learn on this assignment do extend to chess and other games as well! The fundamental question you'll investigate on this assignment is: **using what we've learned about trees, what kinds of Minichess AIs can we design and implement, and how well do they play?**

This assignment is divided into four parts. In Part 0 (not to be handed in), you'll learn about the main data types you'll be using for this program, including a *game tree* that applies what you've in lecture. In Part 1, you'll work with a real-world Minichess data set and build an AI that makes random moves using this dataset as a guide. In Part 2, you'll develop a recursive algorithm for building up a complete game tree, exploring all possible Minichess move sequences up to a given depth, and build a smarter AI that chooses moves based on estimated win probabilities. And in Part 3, you'll develop one final Minichess AI that starts with a completely empty game tree, but that builds up the tree as it plays more and more games, and balances *using learned strategies* from previous games and *exploring new possibilities* by making random choices.

Note: this assignment is challenging for a few main reasons:

1. It contains more starter code across more files than in previous assignments this year, and you'll need to be reading the docstrings carefully to understand how to use them.
2. The domain and algorithms are fairly complex, even though they don't actually require very much code to implement! You'll spend a lot of time *reading instructions* and *drawing pictures* to make sure you understand what's going on, before getting to write any code. (This is a good practice to do in general, but even more necessary than usual on this assignment.)
3. The concepts build on each other from part to part. You'll need to have a solid grasp of what you did on Part 1 to complete Part 2, and understand both parts before completing Part 3. That said, if you focus the bulk of your time completing Parts 1 and 2, you'll find that completing Part 3 takes less time because it involves just a few variations on the previous parts.

Logistics

- Due date: ~~Thursday March 4 before 12pm noon~~ Saturday March 6 before 12pm noon (extended)
- You may work on this assignment individually or with a partner.
- You will submit your assignment solutions on MarkUs (see submission instructions at the end of this handout).
- Please review the Assignment Guidelines and Policies page (https://q.utoronto.ca/courses/196339/pages/policies-and-guidelines-assignments) and the Course Syllabus section on Academic Integrity (https://q.utoronto.ca/courses/196339/assignments/syllabus#academic-integrity).

Starter files

To obtain the starter files for this assignment:

1. Download a2.zip (starter-files/a2.zip).
    - (**Feb 16**) Changed instruction in `a2_game_tree.py` according to Assignment 2 FAQ, General #1 (https://q.utoronto.ca/courses/196339/pages/assignment-2-faq?module_item_id=2314802). If you downloaded the earlier version no need to re-download, just review the FAQ link.
2. Extract the contents of this zip file into your `csc111/assignments/a2` folder. You should *not* create a new subfolder inside `a2`, and instead extract the assignment files directly under `a2`.

There are more starter files than usual, so please note the following:

- The folder `chessboard` can be *completely ignored*. This is a `pygame`-based library used for visualizing games only. You won't be interacting with this library directly.[1]
- `a2_minichess.py` is a Minichess implementation that handles representing the game board, pieces, making moves, and visualizing games. It also contains a demo game run with two "AIs" that play each other by making purely random moves. You will need to understand the *public interface* of this file, but not its implementation.
- `a2_game_tree.py` contains the common `GameTree` data type you'll use and modify throughout this assignment.
- `a2_part1.py`, `a2_part2.py`, and `a2_part3.py` are standard starter files where we've left instructions and space for you to complete your work on this assignment.

General instructions

For this assignment, there are no written questions, and so you will be only submitting in Python files (no LaTeX files or PDFs). See the bottom of this page for Submission instructions.

Problem Domain: Minichess

Before continuing on this assignment handout, please review this page introducing Minichess (minichess.html), which is the simplified version of chess that we'll be studying on this assignment.

Part 0: Representing Minichess in Python

Note: There isn't anything to be handed in for this part! Instead, please read through this section carefully to learn about the main data types you'll be working with for this assignment.

The `MinichessGame` class

In `a2_minichess.py`, we have defined a class called `MinichessGame` that represents the state of a Minichess game.

We can initialize a new game of Minichess as follows:

```
>>> from a2_minichess import MinichessGame
>>> game = MinichessGame()
```

The `MinichessGame` class provides two important methods for making moves that you'll be using on this assignment. We illustrate both of them below.

```
>>> from a2_minichess import MinichessGame
>>> game = MinichessGame()
>>> # Get all valid moves for white at the start of the game.
>>> game.get_valid_moves()
['a2b3', 'b2c3', 'b2a3', 'c2d3', 'c2b3', 'd2c3']
>>> # Make a move. This method mutates the state of the game.
>>> game.make_move('a2b3')
>>> game.get_valid_moves()  # Now, black only has one valid move
['b4b3']
>>> # If you try to make an invalid move, a ValueError is raised.
>>> game.make_move('a4d1')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: Move "a4d1" is not valid
>>> # This move is okay.
>>> game.make_move('b4b3')
```

And finally, if you want to visualize a specific game state, you can call this method:

```
>>> game.get_url()
'https://lichess.org/analysis/standard/8/8/8/8/rqkr4/pPpp4/1PPP4/RQKR4'
```

Try opening that URL in your web browser!

The `Player` abstract class

The `Player` abstract class in `a2_minichess.py` defines the public interface for a Minichess AI that we'll be using throughout this assignment. As you might expect, you'll be defining various subclasses of `Player`:[2]

```python
class Player:
    """An abstract class representing a Minichess AI.

    This class can be subclassed to implement different strategies for playing chess.
    """


    def make_move(self, game: MinichessGame, previous_move: Optional[str]) -> str:
        """Make a move given the current game.

        previous_move is the opponent player's most recent move, or None if no moves
        have been made.

        Preconditions:
            - There is at least one valid move for the given game
        """
        raise NotImplementedError
```

This is a fairly simple interface: given the current game state and the opponent player's most recent move, each subclass must decide what new move to make. Note that the `Player` class itself doesn't keep track of whether it's white or black! Instead, whenever `make_move` is called, the `Player` is making a move *as the current player of the game*.

We've also provided one `Player` subclass called `RandomPlayer`, which makes purely random moves, selecting from all possible valid moves at the current game state. Please review the implementation of `RandomPlayer.make_move` and make sure you understand how it's using `MinichessGame` methods before moving on.

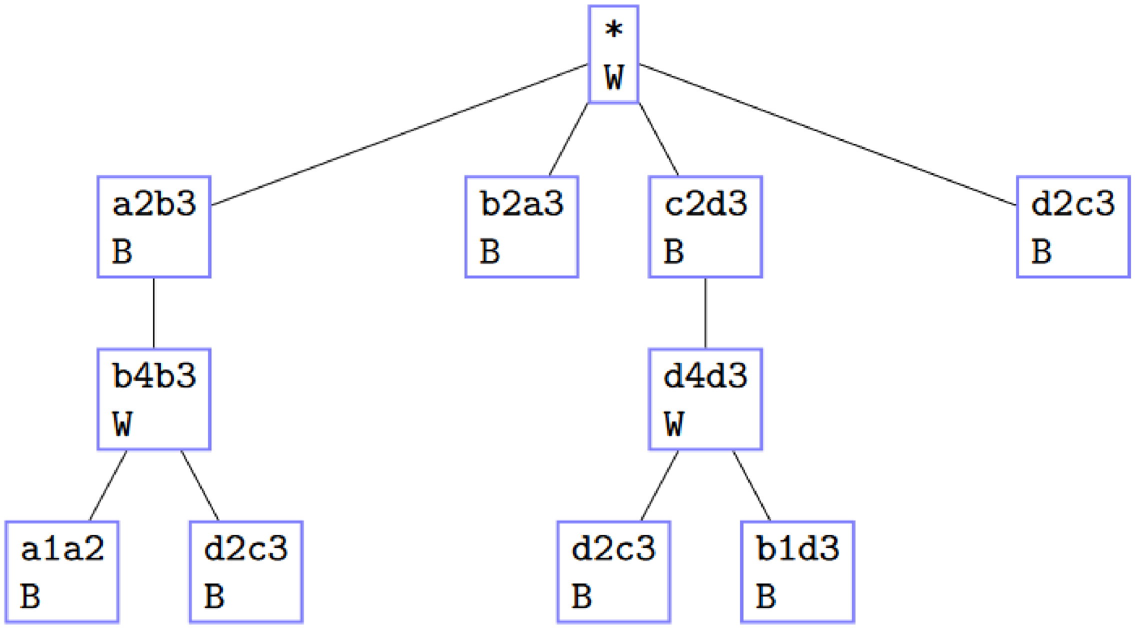
Running a game

`a2_minichess.py` provides two functions for running Minichess games: `run_game`, which runs a single game between two `Player`s, and `run_games`, which allows you to run multiple games. The main block of this file contains two examples of calling `run_games`, and we encourage you to try to run these for yourself as well.

```
if __name__ == '__main__':
    # Demo running Minichess games being played between two random players
    run_games(100, RandomPlayer(), RandomPlayer(), show_stats=True)

    # Try running this to visualize games (takes longer per game)
    # run_games(20, RandomPlayer(), RandomPlayer(), visualize=True,
    #           fps=10, show_stats=True)
```

The `GameTree` class

The previous sections described the code you'll use to represent and run games of Minichess. But the true heart of this assignment is in how you'll use trees to represent a *decision tree of Minichess moves*, encoding information about sequences of moves that each player can make over the course of a game. For example, here is a decision tree showing different possible moves for a game of Minichess.



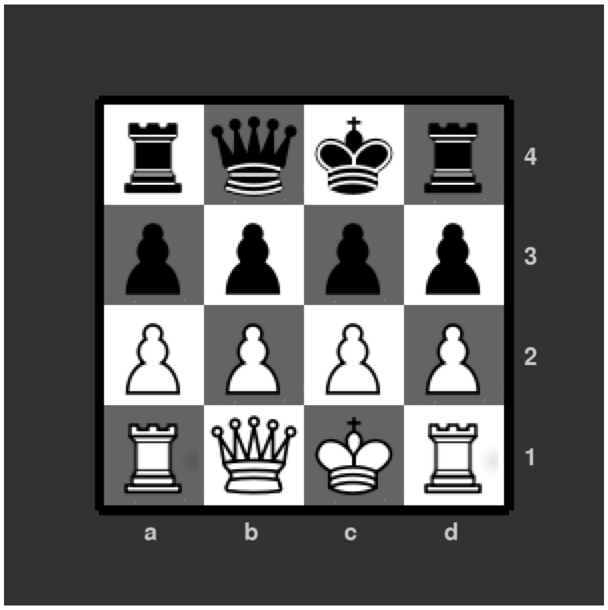
The root of the tree is labelled with `*` to indicate that no moves have been performed yet, and every other node is labelled with a move using the notation described in [our Minichess Overview (minichess.html)](minichess.html). Every node also has either a `W` or `B` indicating the current player—note that this player will make the *next* move, which is represented by the children of that node. Note that *not all possible moves are shown*—when we get to implementing these game trees, we'll find that it is computationally infeasible to create trees showing all possible moves, and instead study ways of creating trees that just contain a "good subset" of the moves.

In `a2_game_tree.py`, the class `GameTree` represents these Minichess decision trees. This class is similar to the `Tree` class from lecture, except that we've replaced the `_root` attribute with a few separate instance attributes storing individual pieces of data, and made those attributes public.[3]

```
class GameTree:
    """A decision tree for Minichess moves.

    Each node in the tree stores a Minichess move and a boolean representing whether
    the current player (who will make the next move) is White or Black.

    Instance Attributes:
      - move: the current chess move (expressed in chess notation), or '*' if this tree
              represents the start of a game
      - is_white_move: True if White is to make the next move after this, False
        otherwise
    """
    move: str
    is_white_move: bool

    # Private Instance Attributes:
    #  - _subtrees: the subtrees of this tree, which represent the game trees after a
        possible
    #               move by the current player
    _subtrees: list[GameTree]
```

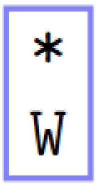
Let's look at an example.

1. At the start of a game of Minichess, the board looks like this:

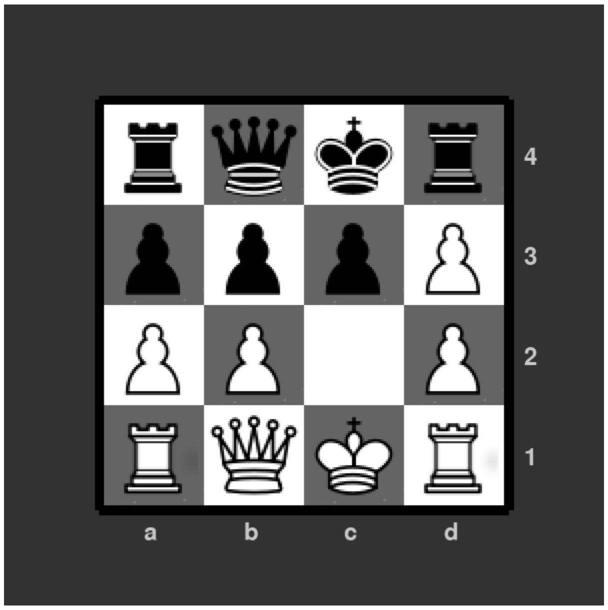
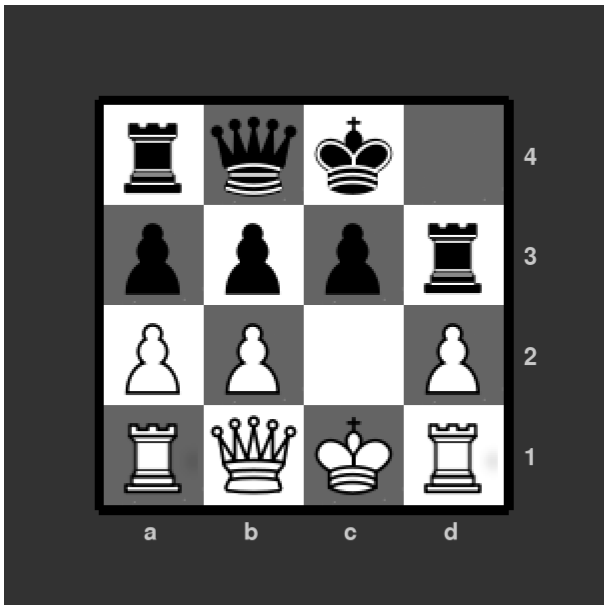


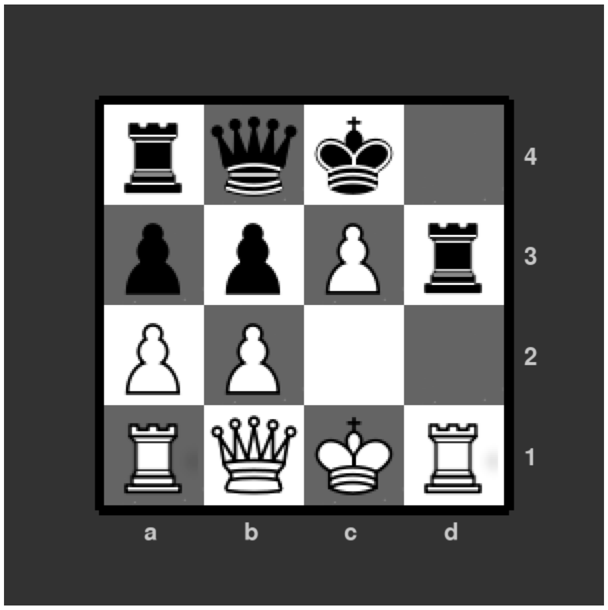
We can represent this starting state with a size-one `GameTree` whose `_move` is `'*'` and whose `is_white_move` is `True` (since it's white's turn to move). What about `_subtrees`? At first we'll say that no moves are recorded by our game tree, and so `_subtrees` is empty.

Visually, we can represent this `GameTree` as the following (note that we show `W/B` instead of `True/False` for clarity in our diagrams).

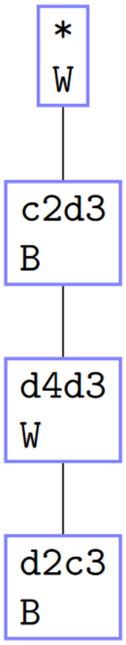


2. Now let's suppose we want to record the following sequence of moves:

Move	**Board state after the move**
`'c2d3'` (white moves its pawn on c2 to d3, capturing a black pawn)	
`'d4d3'` (black moves its rook from d4 to d3, capturing the white pawn)	

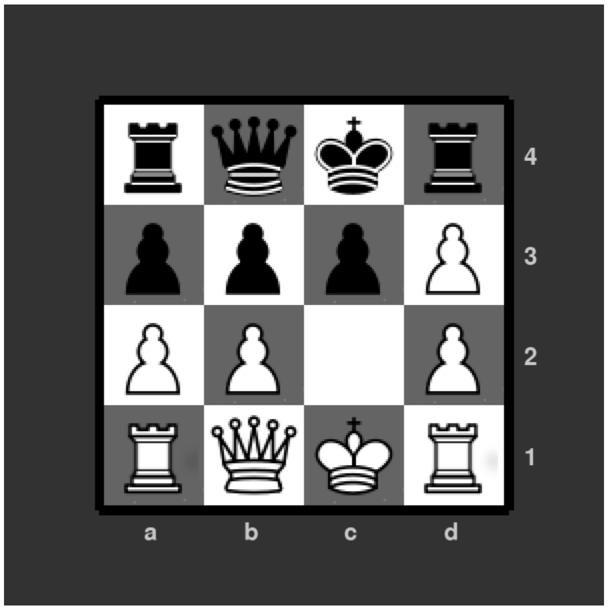
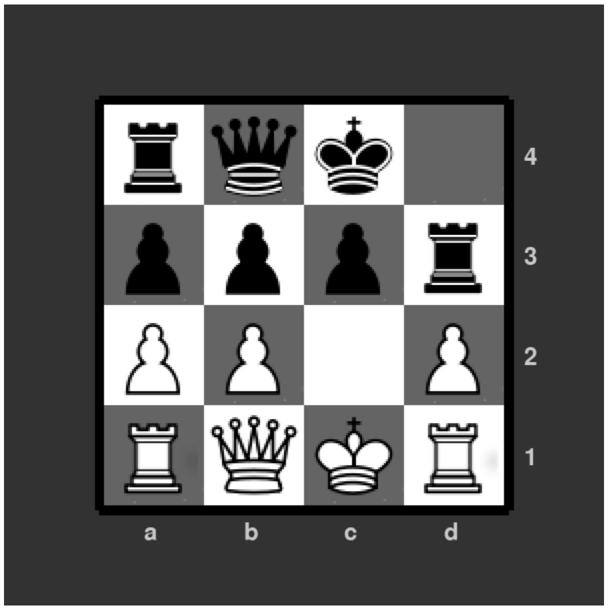
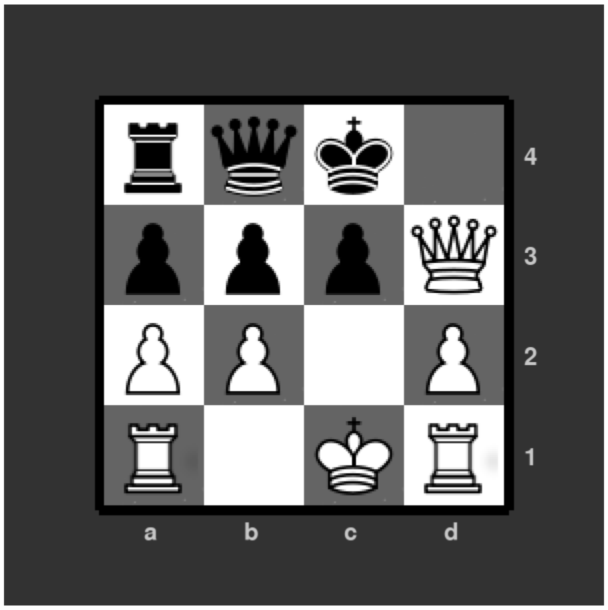
Move	Board state after the move
`'d2c3'` (white moves its pawn on d2 to c3, capturing a black pawn)	

We can do this by adding descendants of our `GameTree`'s root, as follows:

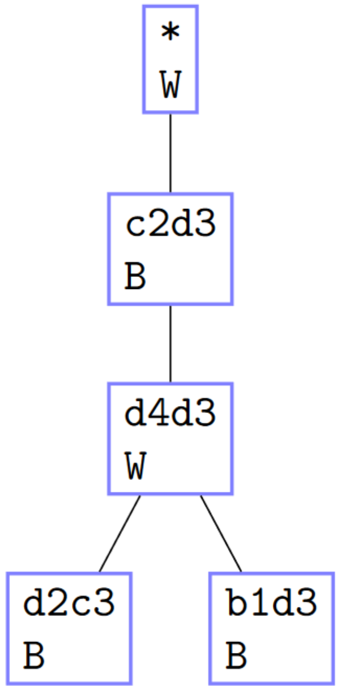


3. Now suppose that we want to record a slightly different move in the third row of the table:

Move	Board state after the move

Move	Board state after the move
`'c2d3'` (white moves its pawn on c2 to d3, capturing a black pawn)	
`'d4d3'` (black moves its rook from d4 to d3, capturing the white pawn)	
(**NEW**) `'b1d3'` (white moves its queen from b1 to d3, capturing the black rook)	

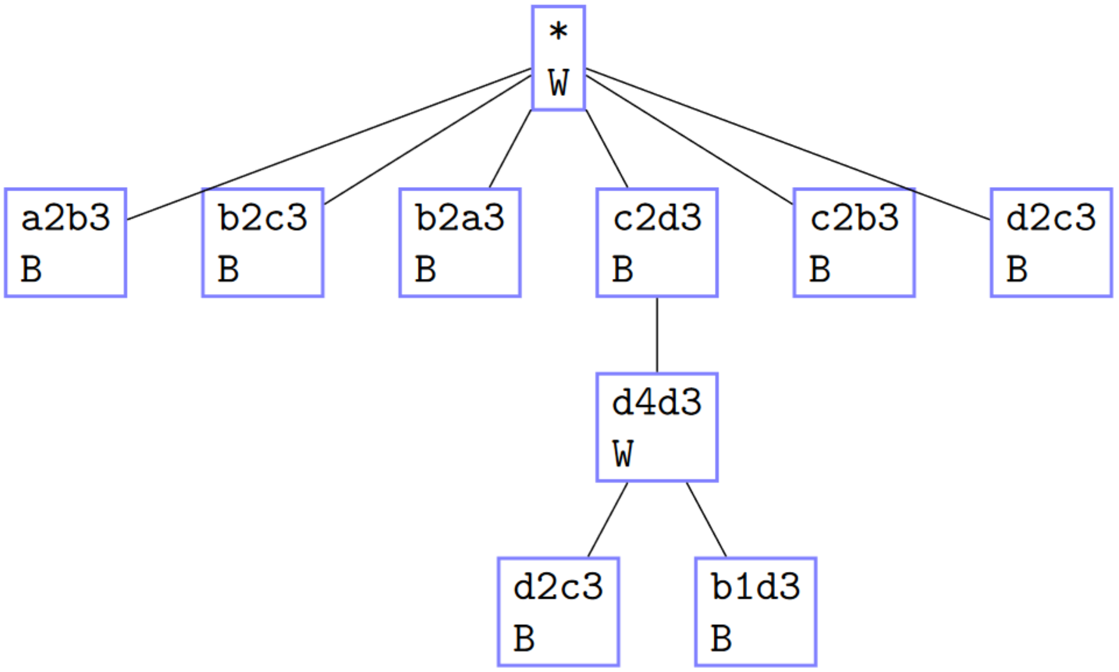
We can add this to our `GameTree` by adding a child of the `'d4d3'` node:



4. And finally, we can use our `GameTree` to add *all possible opening moves* for white. Recall that we can use our `a2_minichess` library to help us:

```
>>> import a2_minichess
>>> game = a2_minichess.MinichessGame()
>>> game.get_valid_moves()
['a2b3', 'b2c3', 'b2a3', 'c2d3', 'c2b3', 'd2c3']
```

So here's how we could update our diagram to show these possible moves (don't worry about the subtree order right now):



Remember, we don't expect you to be able to translate the chess move notation into physical chess moves! You'll be using the `MinichessGame` class to actually compute possible moves and update the state of the game board, so that you don't have to worry about doing this yourself.

You can see an example of manually creating a `GameTree` instance in `a2_part0.py`. Feel free to experiment more in this file, as you aren't handing it in for grading.

Part 1: Loading and "Replaying" Minichess games

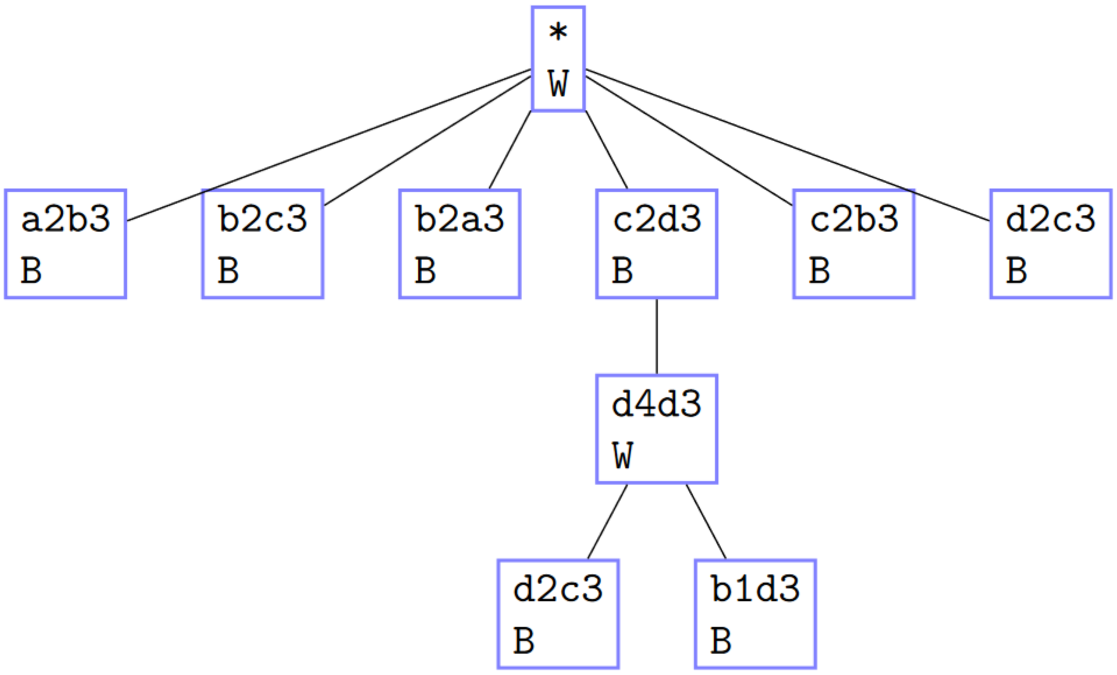
Of course, creating `GameTree`s manually is quite tedious. So your first task for this assignment is to create `GameTree`s by using a "real-world" data set.[4]

1. In `a2_game_tree.py`, implement the method `GameTree.insert_move_sequence`, which takes a list of moves and stores them in the `GameTree`. This is analogous to a `Tree` method you implemented in Tutorial 4, but with a different requirement implementation (see starter file for details).
    - (**Feb 16**) Changed instruction in `a2_game_tree.py` according to Assignment 2 FAQ, General #1 (https://q.utoronto.ca/courses/196339/pages/assignment-2-faq?module_item_id=2314802).
2. In `a2_part1.py`, implement the function `load_game_tree`, which returns a `GameTree` from a **csv file** with the following format:
    - Each line of the file is a sequence of moves, where each move string is separated by a comma.
    - The moves all start from the initial game state, with white making the first move.
    - All move sequences are valid, and do not necessarily end with a player winning.

    For example, given this sample csv contents (also in the starter files under `data/small_sample.csv`):

    ```
    a2b3
    b2c3
    b2a3
    c2d3,d4d3,d2c3
    c2b3
    d2c3
    c2d3,d4d3,b1d3
    ```

    The generated `GameTree` looks like the following:



3. Now let's put our `GameTree`s to use in our Minichess AIs! In `a2_part1.py`, we've created a `Player` subclass called `RandomTreePlayer`, which is initialized with a `GameTree` instance that it uses to make moves.

   When a `RandomTreePlayer` is initialized, its `_game_tree` attribute refers to a `GameTree` representing the initial state of the board (before any moves has been played). This player then keeps track of the moves it makes, and the moves its opponent makes, by descending into the `GameTree`, using code like:[5]

```
self._game_tree = ... self._game_tree ...
```

RandomTreePlayer AI

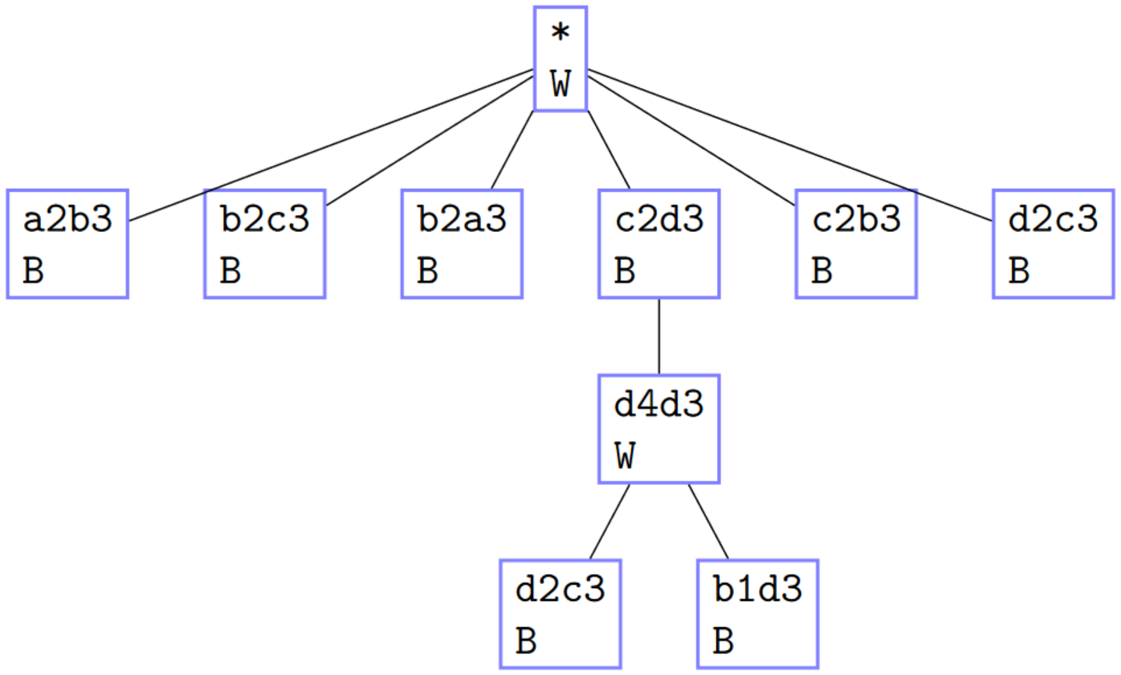
Formally, on a `RandomTreePlayer`'s turn, it does two things:

1. First it updates its game tree to the subtree corresponding to the move made by its opponent. If no subtree is found, its game tree is set to `None`.

2. The player picks its move based on the following conditions:

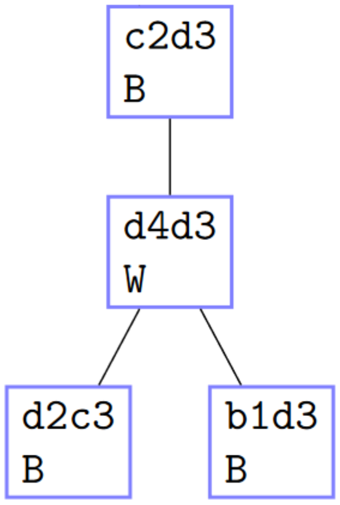
   - If its game tree is `None` or is just a leaf no subtrees, it picks a random move from all valid moves from the game, like `RandomPlayer`.
   - Otherwise, it picks a random move from *among its game tree's subtrees*.

Let's illustrate this with an example. Suppose we create a `RandomTreePlayer` with the following `GameTree`:



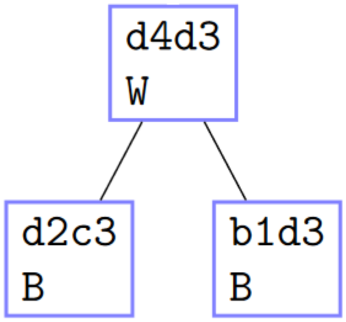
Suppose this `RandomTreePlayer` acts as white, so it goes first.

a. On its first move, its game tree has 6 possible moves to choose from (one for each subtree). It picks one at random. Suppose it picks `'c2d3'`. The player makes that move, and then updates its game tree to be the corresponding subtree:



b. Suppose black makes the move `'d4d3'`.

c. Then on our player's turn, it first updates its game tree to be the subtree corresponding to this move:



And now there are two moves that it can choose from. Suppose it chooses `'b1d3'`, and updates its game tree to be the right subtree.

d. Suppose black makes the move `'c4d3'`.

e. There is no subtree in our player's game tree that corresponds to the opponent's move `'c4d3'`!

   So then the player sets its game tree to `None`, and makes *purely random moves* for the rest of the game.

Note that in this example, we could have reached a `None` game tree much earlier. For example, if the player's opening move had been `'a2b3'`, then no matter what move black played, there wouldn't have been a subtree corresponding to the move, and so the player's game tree would have been set to `None` at the start of their second turn.

Your task is to implement `RandomTreePlayer.make_move` according to the strategy defined above.

4. Finally, let's put our `RandomTreePlayer` to use. The last function you'll implement for Part 1 is `part1_runner`, which creates a game tree based on csv data, and runs a series of Minichess games between two players with the following two possible configurations:

   1. White is a `RandomTreePlayer` with the given game tree, Black is a `RandomPlayer`.
   2. Both White and Black are `RandomTreePlayer`s that use the same game tree.

   In `data/white_wins.csv`, we've curated a record of 2000 games of Minichess where the White player always won. Try calling `part1_runner` with this dataset in the Python console, for both modes. You'll find that the first mode (where the game is played against a `RandomPlayer`), the White player doesn't win very much, but in the second mode (where both players are using the same game tree), White wins 100% of the time! Essentially, Black is forced to make moves constrained by the game tree, and every path down the tree leads to White winning.

   Not very impressive, but it's a start!

Part 2: Complete Game Trees and Win Probabilities

In the previous part, we built `GameTree`s by using datasets of real Minichess games. In this part, we'll explore how we can construct `GameTree`s when we don't have such a dataset available.

1. A conceptually simple way to construct a `GameTree` is to create a *complete* tree, i.e., one that shows all possible moves that each player could make. Unfortunately, this is not computationally feasible: even for the small game of Minichess, the number of possible move sequences is enormous, and would take a very long time for a computer to compute.

   So instead, you'll use a slightly smarter approach: create a complete `GameTree` up to a given depth `d`, which corresponds to all possible move sequences of length at most `d`.[6] This will still allow us to create a game tree by exploring all possible moves, but limit the exploration to a computationally feasible amount.

   So for example, if we start with a `GameTree` with just the initial state of the board, then:

   - A complete game tree with depth 0 would simply contain the starting state.
   - A complete game tree with depth 1 would contain the starting state, and children for all possible initial moves for White.
   - A complete game tree with depth 2 would contain the starting state, children for all possible initial moves for White, and then all possible first moves for Black in response to those moves.

   Open `a2_part2.py`, and implement the function `generate_complete_game_tree`.

2. Next, we want to have our Minichess AI use `GameTree`s in a more sophisticated way than simply making random choices for each move. To do this, we'll need to track how "good" each move is in a `GameTree`.

   For this question, we'll focus on the White player. Your task is to add a new public instance attribute `white_win_probability` to the `GameTree` class, which represents the probability that White will win from the current state of the game, *assuming* that:

   - White always chooses the move that leads to the subtree with the highest win probability.
   - Black always chooses a *random* subtree when making a move.

   To do this, you'll need to make a few changes to the `GameTree` class:

   a. Add the new instance attribute `white_win_probability` (type annotation and documentation!), and add a representation invariant that it is between 0 and 1 inclusive.

b. Add an *optional parameter* to `GameTree.__init__` that allows a user to specify a win probability when creating a new `GameTree`.

   - Your optional parameter must come after the existing parameters.
   - The default value of the parameter is **0.0**.

c. Implement the method `GameTree._update_white_win_probability`, which updates `self.white_win_probability` based on the mathematical definition found in its docstring. Call this method in `GameTree.add_subtree` to update the win probability when a new subtree is added.

d. Add an optional parameter to `GameTree.insert_move_sequence` to specify a win probability when inserting a move sequence *that results in a new leaf being created*.

   - Same instructions as (b) apply.
   - If the given move sequence traces a path that is already entirely in the tree, no win probabilities should be updated, as no new nodes are actually added to the tree.

e. Finally, modify `generate_complete_game_tree` in `a2_part2.py` so that when it creates a new `GameTree`, if the current game state's *winner* is `'White'`, the new `GameTree` is given a white win probability of `1.0`. (In all other cases, we keep the win probability at `0.0`. This is very conservative, as it equates draws with Black wins! But it's the simplest approach for this assignment.)

   You may, but are not required, to modify `load_game_tree` from Part 1.

3. Complete the implementation of `GreedyTreePlayer`, a new `Player` subclass in `a2_part2.py`, according to the following description.

GreedyTreePlayer AI

This player is very similar to `RandomTreePlayer`, except now when `self._game_tree` is not `None` and has at least one subtree:

- If playing as White, the player picks the subtree with the *highest* `white_win_probability`.
- if playing as Black, the player picks the subtree with the *lowest* `white_win_probability`.

If there are ties, pick the *leftmost* subtree with the max/min whte win probability.

4. Finally, complete the function `part2_runner`, which puts together all of your work for this part.

   Try calling this function with depth 5, and between 20-50 games. You should see that White wins a clear majority of the time—much better than the `RandomTreePlayer` from Part 1!

On the other hand, try swapping the players, so White is the `RandomPlayer` and Black is the `GreedyTreePlayer`. Even though the tree is the exact same, Black doesn't do well at all. Remember how we defined the win probabilities: White always makes the "best" move, but Black chooses randomly; and the probabilities assign "Draw" and "Black wins" as the same value (`0.0`), even though these are very different from Black's point of view. *The `white_win_probability` attribute values are biased in favour of White—they aren't as useful to the Black player*.

Part 3: Winning from Nothing

In the previous two parts, we treated *creating a game tree* and *using a game tree in a Minichess AI* as separate phases. However, in artificial intelligence it is common to combine the actions of creating and using a data model, so that the model improves the more it is used. In this final part of our assignment, you'll put this idea into practice.

First, consider one last `Player` subclass called `ExploringPlayer` that uses its `GameTree` to select its moves, but sometimes selects moves randomly so that it explores new moves.

ExploringPlayer AI

An `ExploringPlayer` is initialized with both a game tree and a probability p between 0 and 1 inclusive, called its *exploration probability*.

When the player chooses its move, it behaves exactly like `GreedyTreePlayer`, *except* if its game tree is not `None` and has at least one subtree, the player first generates a random float between 0 and 1 inclusive.

- If the random number is $< p$, the player chooses a random move from among all valid moves for the current game state. If that move doesn't correspond to an existing subtree, its `_game_tree` attribute is set to `None`.
- If the number is $\geq p$, then the player chooses one of its subtrees using the same algorithm as `GreedyTreePlayer`.

1. Your first task is to open `a2_part3.py` and implement this new subclass. You should be able to reuse much of the same code as `GreedyTreePlayer`.

2. Now let's see how we can use this new class to build up a `GameTree` from scratch that gives White an excellent probability of winning.

Your next task is to complete the function `run_learning_algorithm`, which combines our `ExploringPlayer` with a loop that plays more and more games but uses `GameTree.insert_move_sequence` to add the move sequences (and updates white win probabilities) after each game.

The key idea is that *the same game tree* is used across multiple games, and the `ExploringPlayer`s will balance between making known good moves (have high win probabilities) with making random moves to explore and potentially learn new strategies.

3. (*not graded*) You might have wondered why we have `run_learning_algorithm` take in a list of exploration probabilities, rather than just having a fixed probability.

   In `part3_runner` at the bottom of the file, you'll see that we've written a call to `run_learning_algorithm` on a list of length 1000 where every probability is `0.5`. Try running the function—unfortunately, it doesn't look like this algorithm produces a very good player!

   Try replacing the `0.5` with two other extremes:

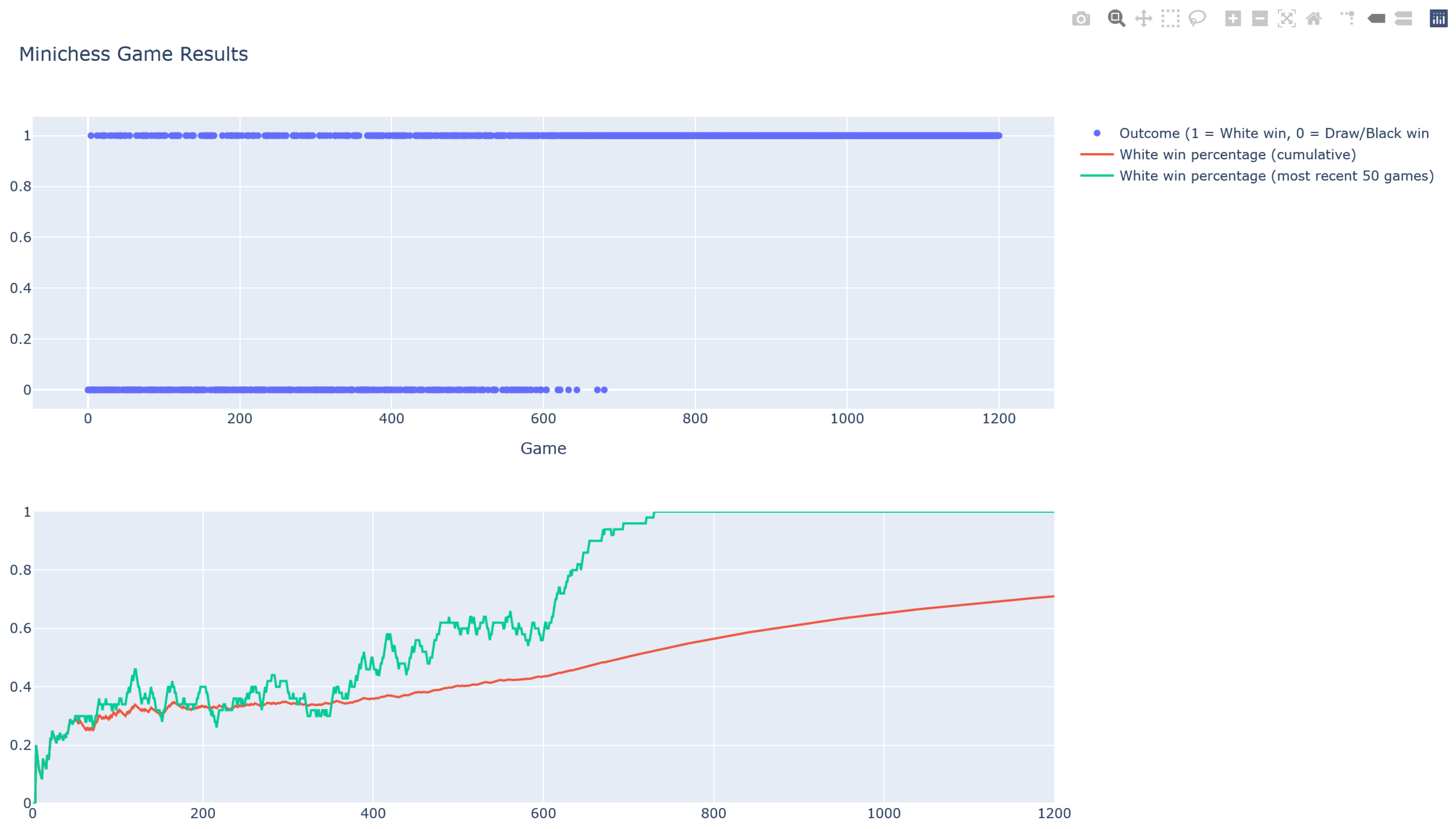
   - `0.0`: in this case, `ExploringPlayer` should behave exactly like `GreedyTreePlayer`. You'll see that even though new games are continually added to the tree, the `ExploringPlayer` doesn't seem to do that well.
   - `1.0`: in this case, `ExploringPlayer` should behave exactly like `RandomTreePlayer`. It doesn't matter that new games are continually added to the tree, as `ExploringPlayer` always makes random moves!

   Here is our intuition: early on when not many games have been played, we want our `ExploringPlayer` to explore a lot, trying out many different moves to try to discover new "strategies" (move sequences). But after having played many games and accumulated meaningful white win probabilities, our `ExploringPlayer` should explore less, so that it can take advantage of good strategies it previously explored.

   - Try making the exploration probabilities a *linear function* that goes from `1.0` down to `0.0` in `n` steps (where `n` is a number you choose, like 1000). With this change alone, you should start seeing a remarkable result: the cumulative probabilities should gradually increase over time.

   - But it turns out that for Minichess, we can do better! Try making your exploration probabilities a *piecewise function*, where the first `n` values are a linear function decreasing from `1.0` to `0.0`, and then next `m` values are constant at `0.0`. The intuition here is that all the exploration happens in the first `n` iterations, and then after that no more exploration needs to occur.

We'll let you experiment with different values of `n` and `m`, and because of randomness you may need to repeat your runs multiple times, but eventually you should find something amazing: after the first `n` games, your `ExploringPlayer` *wins 100% of the time against a* `RandomPlayer` *for the remaining games*! The fact that this behaviour emerges from pure randomness—no dataset, no precomputed complete game tree—is truly extraordinary, and a fitting send-off to our study of trees.



Submission instructions

Please **proofread** and **test** your work carefully before your final submission!

1. Login to [MarkUs (https://markus.teach.cs.toronto.edu/csc111-2021-01)](https://markus.teach.cs.toronto.edu/csc111-2021-01).
2. Go to Assignment 2, and the "Submissions" tab.
3. Submit the following files: `a2_game_tree.py`, `a2_part1.py`, `a2_part2.py`, `a2_part3.py`.

   Please note that MarkUs is picky with filenames, and so your filenames must match these exactly, including using lowercase letters.

Note: for your Python code files, please make sure they run (in the Python console) before submitting them! Code submissions that do not run will receive a grade of *zero* for that part, which we do not want to happen to any of you, so please check your work carefully.

4. Refresh the page, and then *download each file* to make sure you submitted the right version.

Remember, you can submit your files multiple times before the due date. So you can aim to submit your work early, and if you find an error or a place to improve before the due date, you can still make your changes and resubmit your work.

After you've submitted your work, please give yourself a well-deserved pat on the back and go take a rest or do something fun or eat some chocolate!

---

1. Lightly edited from [https://github.com/ahira-justice/chess-board (https://github.com/ahira-justice/chess-board)](https://github.com/ahira-justice/chess-board).↩

2. You might recall that this is very similar to the class structure you saw in [CSC110 Tutorial 10 (https://www.teach.cs.toronto.edu/~csc110y/fall/tutorials/10-abstract-data-types/)](https://www.teach.cs.toronto.edu/~csc110y/fall/tutorials/10-abstract-data-types/).↩

3. Techincally we could have just made `_root` a tuple (or other collection type) of values, but we wanted to choose meaningful attribute names for each component, without defining a separate data type.↩

4. We actually generated some fake data sets for this assignment, as this version Minichess doesn't have a lot of real data. However, the format is quite similar to real chess datasets.↩

5. This is analogous to a linked list `curr = curr.next` update!↩

6. The “at most” is because a given move sequence might result in a checkmate before `d` moves, and this should be contained in the tree.↩