Prototype 3:

The aim for this prototype is to bring together all of the elements created in the previous prototypes and improve them to create a final program. For example, I will use my exploration into sockets form prototype 1 and the chess computer that I created in prototype 2.

This final prototype should include a graphical user interface in the form of a webpage. The interface should be interactive and should be connected to the server, allowing for a full chess game where the computer make moves to be facilitated.

Functionality that the prototype should have:

The prototype should aim to meet all desirable, high priority and essential components of the success criteria. To do this it must meet all of the 31 points laid out in the design specification within the design section.

To do this the move engine must be optimised. Then the code for the client and server API handlers must be created. Then client side code including the Chess\_Board class and DOM methods must be created.

Annotated code screenshots with description:

The first part of this project was to improve the structure of the python project. The previous structure uses many python files in the same folder for easy import statements. The new directory structure has divided the code up logically into different modules. Here is the directory tree.

PS C:\Users\henry\Documents\computing coursework\prototype 6> tree /f

Folder PATH listing for volume OS

Volume serial number is 4ED8-B070

C:.

│   app.py

│   run\_tests\_fast.ps1

│   run\_tests\_slow.ps1

│   schema\_playground.ipynb

│   \_\_init\_\_.py

│

├───.vscode

│       settings.json

│

├───assorted

│   │   chess\_exceptions.py

│   │   general.py

│   │   safe\_hash.py

│   │   \_\_init\_\_.py

│   │

│   └───\_\_pycache\_\_

│           assorted.cpython-310.pyc

│           chess\_exceptions.cpython-310.pyc

│           general.cpython-310.pyc

│           random\_board\_state.cpython-310.pyc

│           safe\_hash.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

├───chess\_functions

│   │   board\_state.py

│   │   pieces.py

│   │   test\_fast\_board\_state.py

│   │   test\_fast\_pieces.py

│   │   test\_fast\_vector.py

│   │   vector.py

│   │   \_\_init\_\_.py

│   │

│   ├───test\_data

│   │   ├───board\_state

│   │   │       color\_in\_check.yaml

│   │   │       game\_over.yaml

│   │   │       generate\_all\_pieces.yaml

│   │   │       generate\_legal\_moves.yaml

│   │   │       generate\_pieces\_of\_color.yaml

│   │   │       piece\_at\_vector.yaml

│   │   │

│   │   ├───pieces

│   │   │       test\_board\_populated.yaml

│   │   │       test\_empty\_board.yaml

│   │   │

│   │   └───vector

│   │           from\_square.yaml

│   │           vector\_add.yaml

│   │           vector\_in\_board.yaml

│   │           vector\_multiply.yaml

│   │           vector\_to\_square.yaml

│   │

│   └───\_\_pycache\_\_

│           assorted.cpython-310.pyc

│           board\_state.cpython-310.pyc

│           pieces.cpython-310.pyc

│           test\_board\_state.cpython-310.pyc

│           test\_fast\_board\_state.cpython-310.pyc

│           test\_fast\_pieces.cpython-310.pyc

│           test\_fast\_vector.cpython-310.pyc

│           test\_pieces.cpython-310.pyc

│           test\_vector.cpython-310.pyc

│           vector.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

├───chess\_game

│   │   console\_chess.py

│   │   game.py

│   │   game\_web.py

│   │   test\_fast\_game.py

│   │   \_\_init\_\_.py

│   │

│   ├───test\_data

│   │   └───test\_save\_and\_restore

│   │           saved\_game.game

│   │

│   └───\_\_pycache\_\_

│           console\_chess.cpython-310.pyc

│           game.cpython-310.pyc

│           game\_web.cpython-310.pyc

│           game\_with\_difficulty.cpython-310.pyc

│           test\_fast\_game.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

├───database

│   │   create\_database.py

│   │   database.db

│   │   handle\_games.py

│   │   models.py

│   │   \_\_init\_\_.py

│   │

│   └───\_\_pycache\_\_

│           create\_database.cpython-310.pyc

│           handle\_games.cpython-310.pyc

│           models.cpython-310.pyc

│           move\_model.cpython-310.pyc

│           move\_model\_schema.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

├───move\_engine

│   │   broken\_minimax\_parallel.py

│   │   cache\_managers.py

│   │   minimax.py

│   │   minimax\_parallel.py

│   │   parallel\_minimax\_testing.py

│   │   test\_slow\_timed\_minimax\_engine.py

│   │   \_\_init\_\_.py

│   │

│   ├───test\_reports

│   │   ├───test\_depth\_2\_vs\_depth\_1

│   │   │       test\_1.csv

│   │   │       test\_1.games

│   │   │

│   │   └───test\_depth\_3\_vs\_depth\_2

│   │           test\_1.csv

│   │           test\_1.games

│   │

│   └───\_\_pycache\_\_

│           cache\_managers.cpython-310.pyc

│           minimax.cpython-310.pyc

│           minimax\_parallel.cpython-310.pyc

│           test\_minimax.cpython-310.pyc

│           test\_slow\_minimax.cpython-310.pyc

│           test\_slow\_timed\_minimax\_engine.cpython-310.pyc

│           test\_slow\_time\_minimax.cpython-310.pyc

│           test\_time\_minimax.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

│

├───schemas

│   │   cache\_item\_schema.py

│   │   socket\_schemas.py

│   │   \_\_init\_\_.py

│   │

│   └───\_\_pycache\_\_

│           cache\_item\_schema.cpython-310.pyc

│           move\_schema.cpython-310.pyc

│           socket\_schemas.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

├───test\_reports

│   ├───test\_bots\_by\_depth

│   │       test\_depth\_2\_vs\_depth\_1.csv

│   │       test\_depth\_3\_vs\_depth\_2.csv

│   │

│   ├───test\_bots\_by\_time

│   │       test\_10s\_timed\_bot\_vs\_2s\_timed\_bot.csv

│   │       test\_10s\_timed\_bot\_vs\_5s\_timed\_bot.csv

│   │       test\_12s\_timed\_bot\_vs\_2s\_timed\_bot.csv

│   │       test\_13s\_timed\_bot\_vs\_5s\_timed\_bot.csv

│   │       test\_14.0s\_timed\_bot\_vs\_10s\_timed\_bot.csv

│   │       test\_15s\_timed\_bot\_vs\_5s\_timed\_bot.csv

│   │       test\_17s\_timed\_bot\_vs\_2s\_timed\_bot.csv

│   │       test\_20s\_timed\_bot\_vs\_10s\_timed\_bot.csv

│   │       test\_20s\_timed\_bot\_vs\_5s\_timed\_bot.csv

│   │       test\_21.0s\_timed\_bot\_vs\_15s\_timed\_bot.csv

│   │       test\_28.0s\_timed\_bot\_vs\_20s\_timed\_bot.csv

│   │       test\_30s\_timed\_bot\_vs\_15s\_timed\_bot.csv

│   │       test\_40s\_timed\_bot\_vs\_10s\_timed\_bot.csv

│   │       test\_40s\_timed\_bot\_vs\_20s\_timed\_bot.csv

│   │       test\_60s\_timed\_bot\_vs\_30s\_timed\_bot.csv

│   │       test\_6s\_timed\_bot\_vs\_2s\_timed\_bot.csv

│   │       test\_7s\_timed\_bot\_vs\_2s\_timed\_bot.csv

│   │       test\_9s\_timed\_bot\_vs\_5s\_timed\_bot.csv

│   │

│   └───test\_depth\_vs\_randotron

│           test\_depth\_1\_num\_1.csv

│           test\_depth\_1\_num\_2.csv

│           test\_depth\_1\_num\_3.csv

│           test\_depth\_2\_num\_1.csv

│           test\_depth\_2\_num\_2.csv

│           test\_depth\_2\_num\_3.csv

│

├───website

│   │   flask\_server.py

│   │   secret\_key.key

│   │   \_\_init\_\_.py

│   │

│   ├───static

│   │       chess\_class.js

│   │       favicon.ico

│   │       initial\_game\_data.js

│   │       initial\_game\_data.json

│   │       main.js

│   │       style.css

│   │       vector.js

│   │

│   ├───templates

│   │       chess\_game.html

│   │

│   └───\_\_pycache\_\_

│           flask\_server.cpython-310.pyc

│           \_\_init\_\_.cpython-310.pyc

│

This has allowed each python module to be self-contained like a class. All objects from all files within the folder can be accessed internally. The \_\_init\_\_.py files dictate which objects will be exported as part of the module.

I will now explain each module in tern and any changes that were made.

Assorted Module:

This module contained general functions, variables and objects that were small and used throughout the whole program.

Here is the \_\_init\_\_.py file for the module which decides which objects are exported by the module:

# this file decides that components from within this folder are exported as part of the assorted module

from .general import ARBITRARILY\_LARGE\_VALUE, cache\_decorator, dev\_print

from .safe\_hash import safe\_hash

from .chess\_exceptions import TimeOutError, InvalidMove, NotUserTurn, NotComputerTurn, UnexplainedErroneousMinimaxResultError

And is the code that for these objects:

General.py

# this file is just a file of short assorted constants and functions that are general in use

# It only contains small functions as I have tried to group large, similar functions logically in there own file

# from functools import lru\_cache

# cache\_decorator = lru\_cache(maxsize=10000)

def cache\_decorator(func): return func

# this is used in the static evaluation and minimax process. It is used to represent infinity in a way that still allows comparrison

ARBITRARILY\_LARGE\_VALUE = 1\_000\_000

# this function is relatively redundant but allows for print statements in debugging

# in later iteration this may be replaced with logging.

# it is useful as it allows for DEBUG print statements without needing to remove them when finished

DEBUG = False

def dev\_print(\*args, \*\*kwargs):

    if DEBUG:

        print(\*args, \*\*kwargs)

In the above file:

* The dev print method was rarely used as I found it easier to comment out print statements that were used in development
* The ARBITRARILY\_LARGE\_VALUE variable is used to represent infinity. This is needed to provide static evaluations of board states that are checkmate or to set the initial values for alpha and beta in the move engine.
* The cache decorator was applied to various function within the Chess Engine module. You can see that ultimately I set the decorator to do nothing as the use of the decorator made to program noticeable slower. This was not at all the expected outcome.

chess\_exceptions.py

# this error is used to cause the timed minimax function

# raising and then catching this error allows for the algorithm to be self-interrupting

class TimeOutError(Exception):

    pass

# this is an exception that allows for the game data to be bound to it

# this allows for the relevant chess game that caused the error to be examined afterwards

# it is a normal exception except the constructor has been modified to save the game data as a property

class \_\_ChessExceptionTemplate\_\_(Exception):

    def \_\_init\_\_(self, \*args, \*\*kwargs) -> None:

        # none if key not present

        self.game = kwargs.pop("game", None)

        super().\_\_init\_\_(\*args, \*\*kwargs)

# these are custom exceptions.

# these are used primarily by the game class

# they contain no logic but have distinct types allowing for targeted error handling

class InvalidMove(\_\_ChessExceptionTemplate\_\_):

    pass

class NotUserTurn(\_\_ChessExceptionTemplate\_\_):

    pass

class NotComputerTurn(\_\_ChessExceptionTemplate\_\_):

    pass

class UnexplainedErroneousMinimaxResultError(\_\_ChessExceptionTemplate\_\_):

    pass

The above exceptions could hold data about a chess game but generally their functionality and logic wasn’t used. The were used as specific names expressions that could be raised and caught using error handling in the Game Manager and Move Engine Modules.

from hashlib import sha256

# this hash function produces an integer hash of a python object

# it is necessary as the in built hash function produces different hashes for the same object,

# this function always produces the same hash for a given object, this allows the hash to be stored in a database for later use

def safe\_hash(item):

        # convert item (usually tuple) to string

    encoded\_string = repr(item).encode("utf-8")

    # produce a hexadecimal string hash of the object

    hex\_hash = sha256(encoded\_string).hexdigest()

    # convert this to an integer

    # int\_hash = int(hex\_hash, 16)

    # return int\_hash

    return hex\_hash

This hash function that uses sha256 was needed as the python hash function produces a smaller hash. In addition the value that the hash function produces for the same input changes as the program is stopped and rerun. This means that if I want to store the hash of a python object in a persistent database, I will need to use the above hash function.

Chess Functions Module (Chess Engine):

Within this module there have been minimal changes. This module was completed and tested in prototype 2.

The main changes were:

* A hash function was used to hash the board state for use in a minimax cache database entry
* The create random board state function was moved to the board state file (to avoid it being repeated elsewhere)

# this function produces a random board state

# this is done by selecting and then implementing a legal move at random up to so many moves

# it is used in unittest

def random\_board\_state(moves: int):

    board\_state = Board\_State()

    # keep trying to get a non over board state

    while True:

        try:

            # iteratively pick a move at random and implement it until the right number of move is reached

            for \_ in range(moves):

                legal\_moves = list(board\_state.generate\_legal\_moves())

                assert len(legal\_moves) > 0

                random\_move = random\_choice(legal\_moves)

                board\_state = board\_state.make\_move(\*random\_move)

        except AssertionError:

            # if no legal moves (over) then try again

            continue

        else:

            return board\_state

The board state module had the following \_\_init\_\_.py file that decided what functions it exports:

# this file decided which function and classes from within this folder should be exported as part of the chess functions module.

from .board\_state import Board\_State, random\_board\_state

from .vector import Vector

from .pieces import PIECE\_TYPES, Piece, Pawn, Knight, Bishop, Rook, King, Queen

Other changes that were made were due to failed tests. These will be explained later.

Move Engine:

This module is responsible for executing an efficient minimax search to determine the computer’s move. I have heavily modified the code to add various optimisations.

Here is the file minimax.py:

# import local modules

# cannot import game as causes circular import, if necessary put in same file

from board\_state import Board\_State

from assorted import ARBITRARILY\_LARGE\_VALUE

from vector import Vector

# my minimax function takes as arguments:

# Board\_State, is\_maximiser, alpha and beta (used for pruning) and check\_extra\_depth (produces better outcome but slower)

# it returns

# score, child, move

def minimax(board\_state: Board\_State, is\_maximizer: bool, depth, alpha, beta, check\_extra\_depth=True):

    # sourcery skip: low-code-quality, remove-unnecessary-else, swap-if-else-branches

    # assume white is maximizer

    # when calling, if give appropriate max min arg

    # base case

    # if over or depth==0 return static evaluation

    over, \_ = board\_state.is\_game\_over\_for\_next\_to\_go()

    if depth == 0 or over:

        # special recursive case 1

        # examine terminal nodes that are check to depth 2 (variable depth)

        # to avoid goose chaises, extra resources are allowed if check not already explored

        if board\_state.color\_in\_check() and check\_extra\_depth and not over:

            # print(f"checking board state {hash(board\_state)} at additional depth due to check")

            return minimax(

                board\_state=board\_state,

                is\_maximizer=is\_maximizer,

                depth=2,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=False

            )

        else:

            # static eval works for game over to

            return board\_state.static\_evaluation(), None, None

    # define variables used to return more that just score (move and child)

    best\_child\_game\_state: Board\_State | None = None

    best\_move\_vector: Vector | None = None

    # function yields move ordered by how favorable they are (low depth minimax approximation)

    def gen\_ordered\_child\_game\_states():

        # this function does a low depth minimax recursive call (special recursive case 2) to give a move a score

        def approx\_score\_move(move):

            child\_game\_state = board\_state.make\_move(\*move)

            return minimax(

                board\_state=child\_game\_state,

                depth=depth-2,

                is\_maximizer=not is\_maximizer,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=False

            )[0]

            # print(f"approx\_score\_move(move={move!r})  ->  {result!r}")

        # if depth is 1 or less just yield moves form legal moves

        if depth <= 1 :

            yield from board\_state.generate\_legal\_moves()

        # else sort them

        else:

            # sort best to worse

            # sort ascending order if minimizer, descending if maximizer

            yield from sorted(

                board\_state.generate\_legal\_moves(),

                key=approx\_score\_move,

                reverse=is\_maximizer

            )

    if is\_maximizer:

        # set max to -infinity

        maximum\_evaluation = (-1)\*ARBITRARILY\_LARGE\_VALUE

        # iterate through moves and resulting game states

        for position\_vector, movement\_vector in gen\_ordered\_child\_game\_states():

            child\_game\_state = board\_state.make\_move(from\_position\_vector=position\_vector, movement\_vector=movement\_vector)

            # evaluate each one

            # general recursive case 1

            evaluation, \_, \_ = minimax(

                board\_state=child\_game\_state,

                is\_maximizer=not is\_maximizer,

                depth=depth-1,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=check\_extra\_depth

            )

            # update alpha and max evaluation

            if evaluation > maximum\_evaluation:

                maximum\_evaluation = evaluation

                best\_child\_game\_state = child\_game\_state

                best\_move\_vector = (position\_vector, movement\_vector)

                alpha = max(alpha, evaluation)

            # where possible, prune

            if beta <= alpha:

                # print("Pruning!")

                break

        # once out of loop, return result

        return maximum\_evaluation, best\_child\_game\_state, best\_move\_vector

    else:

        minimum\_evaluation = ARBITRARILY\_LARGE\_VALUE

        for position\_vector, movement\_vector in gen\_ordered\_child\_game\_states():

            child\_game\_state = board\_state.make\_move(from\_position\_vector=position\_vector, movement\_vector=movement\_vector)

            evaluation, \_, \_ = minimax(

                board\_state=child\_game\_state,

                is\_maximizer=not is\_maximizer,

                depth=depth-1,

                alpha=alpha,

                beta=beta,

                check\_extra\_depth=check\_extra\_depth

            )

            if evaluation < minimum\_evaluation:

                minimum\_evaluation = evaluation

                best\_child\_game\_state = child\_game\_state

                best\_move\_vector = (position\_vector, movement\_vector)

                beta = min(beta, evaluation)

            if beta <= alpha:

                # print("Pruning!")

                break

        return minimum\_evaluation, best\_child\_game\_state, best\_move\_vector

I have implemented decomposed the large minimax function into sub function within a class. Some parameters are provided directly to the minimax method within the class while others that configure the move engine (e.g. is cache allowed) are instead parameters of the objects constructor. This has helped reduce the parameters to the minimax method which helps avoid unnecessary complexity.

To use the minimax algorithm, the object is initialised and then called like a function. Calling the object in this way runs the \_\_call\_\_ method which runs the minimax\_first\_call method. This method is run only the first time the minimax function is called (not on any recursive calls). The first call handler function is responsible for starting up the cache manager object and using the validator function to check the minimax output if enables. The cache handler objects allow for minimax data to be cached to improve the functions efficiency. Some of the objects has a start up and cool down sequence (e.g. opening and then disposing of a database session). The use of a context manager vie the “with” command calls the \_\_enter\_\_() method on the cache manager object, then completed the minimax search and then calls the \_\_exit\_\_() method on the object.

The method names minimax if responsible for the bulk of the logic behind the minimax algorithm in this implementation although some large blocks of logic have been extracted into separate private methods.

The core principles behind the algorithm remain the same. It has a base case to stop recurring and evaluate a terminal node. A minimax search of a given Board\_State object at a depth of N then uses the recursive case. It iterates though the legal moves of the board and then recursively evaluates the child nodes with a depth of n-1. It then updates its pruning variables and its best variables. Once it stops iterating it returns the best move and score.

In the above algorithm, the base case has been abstracted out into another function called pseudo\_base\_case(). This method will normally returns a static evaluation of the board (base case) but if check has been encountered, it can search to an additional depth using a hidden recursive case. This optimisation should improve the quality of the result.

Minimax method also extracts out the logic of getting the legal moves and child board states to another method called generate\_move\_child. This method uses depth N-2 searches (another hidden recursive case) to estimate the score of each of the child nodes before they are searched. This allows it to sort the child nodes in order from best to worst. This should improve the frequency by which parts of the tree can be pruned. In addition, if a depth N-1 search has already been completed and cached, then this sorting will not cost any additional computations.

The only other change is that the minimax method checks the cache first to see if it can retrieve the result and skip the rest of the function. Then if this wasn’t possible, at the end of the function, the search result is added to the cache. Recursive sub-searches should also be able to retrieve and save results to and from the cache. This means that even if the cache doesn’t already contain the whole search, it may still contain part of it. This will save computations and improve average time complexity at the cost of a worse space complexity.

I then created a series objects that were designed to allow me to easily cache the minimax search results. Each object has the same interface meaning that they can be used in the same way. All logic around how it stores the results is self-contained and abstracted.

Cache\_managers.py

# import from external modules

import os

import json

from time import perf\_counter

# import local modules

from assorted import safe\_hash

from chess\_functions import Board\_State, Vector

from database import create\_session, end\_session, persistent\_DB\_engine, volatile\_RAM\_engine, Minimax\_Cache\_Item

from schemas import minimax\_cache\_item\_schema

# create an blank object with no logic that can be used with a context manager

class Blank\_Context():

    def \_\_enter\_\_(self, \*args, \*\*kwargs):

    # def \_\_exit\_\_(self, exception\_type, exception\_value, traceback):

        pass

    def \_\_exit\_\_(self, \*args, \*\*kwargs):

        pass

# ram cache doesn't need access to a context manager so it uses the blank one

class RAM\_cache(Blank\_Context):

    # cache is stored in a local dictionary property in the ram cache object

    def \_\_init\_\_(self) -> None:

        self.memoization\_cache = dict()

    def cache\_size(self):

        return len(self.memoization\_cache)

    # the result of a minimax call is added to the cache

    def add\_to\_cache(self, board\_state: Board\_State, depth, score, move):

        # moves are stored as tuples of integers

        def serialize\_move(move):

            if move is None:

                return None

            return (

                (move[0].i, move[0].j),

                (move[1].i, move[1].j)

            )

        # ensure that the cache exists in the memoization\_cache property

        assert self.memoization\_cache is not None

        # can use a board\_state as a key as it is hashable

        # the board state hash is used as the key within the dictionary (which acts as a hash table)

        board\_state\_hash = board\_state.database\_hash()

        # print(f"CALL: add\_to\_cache(self, board\_state={hash(board\_state)} depth={depth}, score={score}, move={move}")

        # print(f"self.memoization\_cache.get(board\_state\_hash) is None    -->    {self.memoization\_cache.get(board\_state\_hash) is None}")

        # decide if the cache should be updated based on whether the record already existed and if so, its depth

        if self.memoization\_cache.get(board\_state\_hash) is None:

            needs\_update = True

        else:

            best\_depth\_in\_cache = self.memoization\_cache[board\_state\_hash]["depth"]

            # print(f"best\_depth\_in\_cache < depth    -->    {best\_depth\_in\_cache} < {depth}   --->   {best\_depth\_in\_cache < depth}")

            needs\_update = best\_depth\_in\_cache < depth

        # print(f"Cache:  {self.memoization\_cache}")

        # if need an update then add the record (lesser depth cache overwritten)

        if needs\_update:

            new\_data\_item = {

                "depth": depth,

                "score": score,

                "move": serialize\_move(move)

            }

            self.memoization\_cache[board\_state\_hash] = new\_data\_item

            # if depth >= 1:

            # print(f"New data {board\_state\_hash} added to cache:   {new\_data\_item}")

            # print(f"new\_cache\_size:   {self.cache\_size()}")

        # print(f"Cache needed updating:   {needs\_update}")

    # this function retrieves an item form cache

    def search\_cache(self, board\_state, depth):

        # move is converted from tuple of integers back to vectors when deserialized

        def deserialize\_move(serialised\_move):

            if serialised\_move is None:

                return None

            return (

                Vector(\*serialised\_move[0]),

                Vector(\*serialised\_move[1])

            )

        assert self.memoization\_cache is not None

        # get the hash ov hte board state

        # board\_state\_hash = hash(board\_state)

        board\_state\_hash = board\_state.database\_hash()

        # if not in cache, return none

        if self.memoization\_cache.get(board\_state\_hash) is None:

            # if depth >= 2:

            #     print(f"item {board\_state\_hash} not in cache (depth={depth})")

            return None

        # else check the depth is adequate and return if appropriate

        else:

            best\_depth\_in\_cache = self.memoization\_cache[board\_state\_hash]["depth"]

            record\_useful = best\_depth\_in\_cache >= depth

            if record\_useful:

                # if depth >= 2:

                # print(f"RAM\_Cache used for {board\_state\_hash}:   {self.memoization\_cache[board\_state\_hash]}")

                data\_item = self.memoization\_cache[board\_state\_hash].copy()

                data\_item["move"] = deserialize\_move(data\_item["move"])

                return data\_item

            else:

                # if depth >= 2:

                    # print(f"cache not useful for item {board\_state\_hash}:  {self.memoization\_cache[board\_state\_hash]}, required depth is {depth}")

                # print(f"cache searched but not used")

                return None

# this function creates a persistent cache in a json file

class JSON\_Cache(RAM\_cache):

    # when the object is created, load the contents of the json file into the memoization cache variable

    def \_\_init\_\_(self, file\_path=None) -> None:

        # print("Initializing json cache")

        if file\_path is not None:

            self.file\_path = file\_path

        else:

            self.file\_path = r"./database/minimax\_cache.json"

        # default

        self.memoization\_cache = {}

        # try get data from file

        if os.path.exists(self.file\_path):

            with open(self.file\_path, "r") as file:

                content = file.read()

            try:

                assert content != "", "Json file was blank / empty"

                cache\_data = json.loads(content)

                # fix that in json, board state hash key is sting

                new\_items = map(

                    lambda item: (int(item[0]), item[1]),

                    cache\_data.items()

                )

                cache\_data = dict(tuple(new\_items))

                self.memoization\_cache = cache\_data

            except Exception as e:

                # print(e)

                # print("Failed to load json cache data so using blank cache")

                pass

    # inherited ram cache method add to and retrieve cache form the memoization cache variable

    # write the memoization cache dictionary to a json file

    def save(self):

        print(f"Saving cache (size={self.cache\_size()}) to json file")

        with open(self.file\_path, "w") as file:

            file.write(

                json.dumps(

                    self.memoization\_cache

                )

            )

    # do this when the cache manager is closed by the context manager

    def \_\_exit\_\_(self, \*args, \*\*kwargs):

        self.save()

# this json cache doesn't save automatically

class JSON\_Cache\_Manual\_Save(JSON\_Cache):

    def \_\_exit\_\_(self, \*args, \*\*kwargs):

        pass

# database cache connects to the database to store minimax cached results

# it uses a database in RAM for small cache and a persistent database for searches at a greater depth

class DB\_Cache():

    # by default, depth 0 uses ram cache and greater depth uses persistent cache

    def \_\_init\_\_(self, min\_DB\_depth=1):

        # set the min depth parameter as a property and then define other properties with starting values

        self.min\_DB\_depth = min\_DB\_depth

        self.\_RAM\_cache = None

        self.\_DB\_session = None

        self.engaged = False

        # a cached depth 3 call can be used if a depth 2 call is needed

        self.allow\_greater\_depth = True

        self.time\_delta\_DB = 0

        self.time\_delta\_schemas = 0

        self.checks\_to\_cache = 0

        self.retrieved\_item = 0

        self.items\_added = 0

    def \_\_enter\_\_(self, \*args, \*\*kwargs):

        # when the context manager is used and the boot up sequence runs (this method)

        # a database session is created (using the database module)

        # assert all(e is None for e in (self.\_RAM\_cache, self.\_DB\_session))

        assert not self.engaged, "Must not be engaged (entered) already to enter"

        self.engaged = True

        # these variables keep track of various data points while the cache is used within a context manager

        self.time\_delta\_DB = 0

        self.time\_delta\_schemas = 0

        self.checks\_to\_cache = 0

        self.retrieved\_item = 0

        self.items\_added = 0

        # this allows for the cache manager to be entered multiple times on accident without making multiple sessions

        # self.\_RAM\_cache = RAM\_cache()

        # self.\_persistent\_DB\_session = scoped\_session(sessionmaker(bind=persistent\_DB\_engine))

        # self.\_volatile\_RAM\_session = scoped\_session(sessionmaker(bind=volatile\_RAM\_engine))

        self.\_persistent\_DB\_session = create\_session(persistent\_DB\_engine)

        self.\_volatile\_RAM\_session = create\_session(volatile\_RAM\_engine)

    def \_\_exit\_\_(self, \*args, \*\*kwargs):

        # when the context manager runs the close function of the cache manager (this method)

        # the session objects are closed and discarded and tracker variables are set back to 0

        assert self.engaged, "Must be engaged (entered) to exit"

        self.engaged = False

        end\_session(self.\_persistent\_DB\_session)

        self.\_persistent\_DB\_session = None

        end\_session(self.\_volatile\_RAM\_session)

        self.\_volatile\_RAM\_session = None

        # print(f"Cache session lasted {round(self.time\_delta\_DB, 2)}+{round(self.time\_delta\_schemas, 2)} sec (DB + schema time): {self.checks\_to\_cache} checks made, {self.retrieved\_item} items retrieved, {self.items\_added} items added")

        self.time\_delta\_DB = 0

        self.time\_delta\_schemas = 0

        self.checks\_to\_cache = 0

        self.retrieved\_item = 0

        self.items\_added = 0

    def \_search\_DB\_cache(self, session, board\_state, depth):

        # this function hashes the board state and searches the database for cache corresponding to that hash

        # board\_state\_hash = str(hash(board\_state))

        board\_state\_hash = board\_state.database\_hash()

        start = perf\_counter()

        # cache must have a matching hash and a sufficient depth

        # ORM allows for queries to be written in a pythonic way

        if self.allow\_greater\_depth:

            result = session.query(Minimax\_Cache\_Item)\

                .where(

                    Minimax\_Cache\_Item.board\_state\_hash == board\_state\_hash

            )\

                .where(

                    Minimax\_Cache\_Item.depth >= depth

            )\

                .first()

        else:

            result = session.query(Minimax\_Cache\_Item)\

                .where(

                    Minimax\_Cache\_Item.board\_state\_hash == board\_state\_hash

            )\

                .where(

                    Minimax\_Cache\_Item.depth == depth

            )\

                .first()

        stop = perf\_counter()

        self.time\_delta\_schemas += stop - start

        # if not item found then return none

        if result is None:

            return None

        start = perf\_counter()

        # use the schema object to deserialize the cached item before returning it.

        result = minimax\_cache\_item\_schema.dump(result)

        stop = perf\_counter()

        self.time\_delta\_schemas += stop - start

        return result

    def \_add\_to\_DB\_cache(self, session, board\_state: Board\_State, score, depth, move):

        # this function adds an item to the cache

        start = perf\_counter()

        # board state is hashed

        # print("adding item to database cache")

        # board\_state\_hash = str(hash(board\_state))

        board\_state\_hash = board\_state.database\_hash()

        # if depth >= 2:

        #     print(f"Adding item {board\_state\_hash} at depth {depth} to cache")

        # new database entry item created with schema

        new\_item = minimax\_cache\_item\_schema.load(

            dict(

                board\_state\_hash=board\_state\_hash,

                score=score,

                depth=depth,

                move=move

            )

        )

        stop = perf\_counter()

        self.time\_delta\_schemas += stop - start

        start = perf\_counter()

        # other searches at a lesses depth are deleted

        # (assumes cache already checked and the other cache is worse)

        session.query(Minimax\_Cache\_Item)\

            .where(

            Minimax\_Cache\_Item.board\_state\_hash == board\_state\_hash

        )\

            .delete()

        # print(f"move   -->   {move!r}")

        # print(f"New item:   {new\_item!r}")

        # the item is added to the database and committed

        session.add(new\_item)

        session.commit()

        stop = perf\_counter()

        self.time\_delta\_DB += stop - start

    def search\_cache(self, board\_state: Board\_State, depth):

        # this function searches the cache

        assert self.engaged, "Context manager must be used"

        self.checks\_to\_cache += 1

        # not sure why depth sometimes is a single length tuple containing an int

        # here is a quick fix

        if isinstance(depth, tuple):

            if len(depth) == 1:

                depth = depth[0]

        # print(f"(depth, self.min\_DB\_depth)     -->     {(depth, self.min\_DB\_depth)}")

        # decide which session to use based on depth

        if depth < self.min\_DB\_depth:

            session = self.\_volatile\_RAM\_session

            # return self.\_RAM\_cache.search\_cache(board\_state=board\_state, depth=depth)

        else:

            session = self.\_persistent\_DB\_session

            # return self.\_search\_DB\_cache(board\_state=board\_state, depth=depth)

        # search the cache with this session and then return the result

        result = self.\_search\_DB\_cache(session=session, board\_state=board\_state, depth=depth)

        if result is not None:

            self.retrieved\_item += 1

        return result

    def add\_to\_cache(self, board\_state: Board\_State, score, depth, move):

        # this function adds to the cache

        # this function assumes that no move valuable cache already exists (it overwrites all other cache)

        assert self.engaged, "Context manager must be used"

        # this is used to tackle any bugs elsewhere in the program

        # it ensures that no invalid cache items are added to the database

        def absolute(x): return x if x >= 0 else 0-x

        if absolute(score) > 1\_000\_000 or (move is None and depth > 0):

            # don't add erroneous data to cache

            return None

        # decides that session to use based on depth

        if depth < self.min\_DB\_depth:

            session = self.\_volatile\_RAM\_session

            # self.\_RAM\_cache.add\_to\_cache(board\_state=board\_state, score=score, depth=depth, move=move)

        else:

            session = self.\_persistent\_DB\_session

            # self.\_add\_to\_DB\_cache(board\_state=board\_state, score=score, depth=depth, move=move)

        # the caches item is added to the database

        self.\_add\_to\_DB\_cache(session=session, board\_state=board\_state, score=score, depth=depth, move=move)

        self.items\_added += 1

    # the hash function describes only the data that makes this cache object unique

    def \_\_hash\_\_(self) -> int:

        hash(safe\_hash(

            (

                "DB\_Cache",

                # self.\_RAM\_cache,

                # self.\_DB\_session,

                self.min\_DB\_depth,

                self.engaged,

                self.allow\_greater\_depth,

                # self.time\_delta\_DB

                # self.time\_delta\_schemas

                # self.checks\_to\_cache

                # self.retrieved\_item

                # self.items\_added

            )

        ))

Here we can see that I created a series of increasingly complex cache managers. I wanted to have a form of persistent cache as this would allow the program to learn. I decided to use a database rather than a json file for this as a database is more scalable and has better look up times. In addition, a database can be accessed simultaneously by many concurrent processes whereas a json file cannot.

I then created some more efficient versions of the matrix algorithm that inherited from the Move Engine Class.

Minimax\_parallel.py

# import external and local modules

# import sys

import multiprocessing

from time import perf\_counter

from .minimax import Move\_Engine, is\_time\_expired

from .cache\_managers import RAM\_cache, DB\_Cache

from chess\_functions import Board\_State

from assorted import ARBITRARILY\_LARGE\_VALUE, TimeOutError, UnexplainedErroneousMinimaxResultError

# import time

def get\_cores():

    return multiprocessing.cpu\_count()

    # return multiprocessing.cpu\_count()//2

# Classes with the name of JOB are essentially functions that represent a general task that can be completed concurrently

# they are classes as functions cannot be easily piped between threads

# this object represents a job that a concurrent worker should complete,

# I used an object as a function cannot be sent between workers

class Minimax\_Sub\_Job:

    # contractor: configure bot by setting these parameters as properties

    def \_\_init\_\_(self, board\_state, depth, args, kwargs, additional\_depth, cache\_allowed, max\_time=None) -> None:

        # create a cache manager as necessary

        self.cache\_allowed = cache\_allowed

        self.cache\_manager = DB\_Cache() if cache\_allowed else None

        # create a move engine as necessary

        self.move\_engine: Move\_Engine = Move\_Engine(

            cache\_manager=self.cache\_manager,

            cache\_allowed=cache\_allowed,

            additional\_depth=additional\_depth,

        )

        # assert isinstance(self.move\_engine, Move\_Engine), f"Minimax\_Sub\_Job.\_\_init\_\_    self.move\_engine of unexpected type {type(self.move\_engine)}\n{self.move\_engine!r}"

        # assign parameters as properties

        self.board\_state: Board\_State = board\_state

        self.depth = depth

        self.max\_time = max\_time

        # these are other erroneous argument that could be passed to the minimax call

        self.args = args

        self.kwargs = kwargs

    def \_\_call\_\_(self, legal\_moves\_sub\_array):

        # perform part of a minimax search using a sub set of legal moves

        # print(f"running minimax job with legal moves sub array: (hash={hash(str(legal\_moves\_sub\_array))})")

        # print(f"STARTING sub job on moves segment:   {hash(str(legal\_moves\_sub\_array))}")

        # print(legal\_moves\_sub\_array)d\_state,

        # result = self.move\_engine.minimax(

        # assert isinstance(self.move\_engine, Move\_Engine), f"Minimax\_Sub\_Job.\_\_call\_\_    self.move\_engine of unexpected type {type(self.move\_engine)}\n{self.move\_engine!r}"

        if self.cache\_allowed:

            with self.cache\_manager:

                result = self.move\_engine.minimax(

                    board\_state=self.board\_state,

                    depth=self.depth,

                    legal\_moves\_to\_examine=legal\_moves\_sub\_array,

                    # is\_time\_expired = self.is\_time\_expired,

                    max\_time=self.max\_time,

                    \*self.args,

                    \*\*self.kwargs

                )

        else:

            result = self.move\_engine.minimax(

                board\_state=self.board\_state,

                depth=self.depth,

                legal\_moves\_to\_examine=legal\_moves\_sub\_array,

                # is\_time\_expired = self.is\_time\_expired,

                max\_time=self.max\_time,

                \*self.args,

                \*\*self.kwargs

            )

      # print(f"FINISHING sub job on moves segment:   {hash(str(legal\_moves\_sub\_array))}")

        # print(f"Minimax sub job finished (hash={hash(str(legal\_moves\_sub\_array))})")

        return result

# this bot is responsible for performing a low depth search of a move to be used to presort the moves

class Presort\_Moves\_Sub\_Job:

    # construct object

    def \_\_init\_\_(self, depth, board\_state: Board\_State, cache\_allowed, max\_time=None) -> None:

    # def \_\_init\_\_(self, depth, board\_state: Board\_State, move\_engine: Move\_Engine, cache\_allowed, cache\_manager, max\_time=None) -> None:

        # cache manager parameter may not be needed if the cache manager object is also bound to the move engine

        # assign parameters as properties

        self.depth = depth

        self.board\_state = board\_state

        # create a cache manager as necessary

        self.cache\_allowed = cache\_allowed

        self.cache\_manager = DB\_Cache() if cache\_allowed else None

        # create a move engine object

        self.move\_engine = Move\_Engine(

            presort\_moves=True,

            additional\_depth=0,

            cache\_allowed=self.cache\_allowed,

            cache\_manager=self.cache\_manager,

            # cache\_allowed=self.cache\_allowed,

            # cache\_manager=self.cache\_manager,

        )

        # self.cache\_allowed = cache\_allowed

        # self.cache\_manager = cache\_manager

        self.is\_time\_expired = is\_time\_expired

        self.max\_time = max\_time

    def \_\_call\_\_(self, move):

        # cache manager already built into the minimax move engine first call method

        # print(f"Presort job using cache manager:   {self.cache\_manager!r}")

        # with self.cache\_manager:

        #     child = self.board\_state.make\_move(\*move)

        #     child\_score, \_ = self.move\_engine.minimax\_first\_call(

        #         board\_state = child,

        #         depth = self.depth

        #     )

        #     # print("Closing cache")

        #     result = (child\_score, move)

        # return result

        # self interrupting

        if is\_time\_expired(self.max\_time):

            raise TimeOutError()

        # score a child, use minimax first call to add cache manager being opened twice

        # no need to say no parallel as a basic move engine is used

        child = self.board\_state.make\_move(\*move)

        if self.cache\_allowed:

            with self.cache\_manager:

                # child\_score, \_ = self.move\_engine.minimax\_first\_call(

                child\_score, \_ = self.move\_engine.minimax(

                    board\_state=child,

                    depth=self.depth,

                    # is\_time\_expired = self.is\_time\_expired,

                    max\_time=self.max\_time,

                )

        else:

            # child\_score, \_ = self.move\_engine.minimax\_first\_call(

            child\_score, \_ = self.move\_engine.minimax(

                board\_state=child,

                depth=self.depth,

                # is\_time\_expired = self.is\_time\_expired,

                max\_time=self.max\_time,

            )

        return (child\_score, move)

# this is the main class to complete a minimax search in parallel.

# some methods inherited form parent class while some are overwritten

class Parallel\_Move\_Engine(Move\_Engine):

    # construct the engine using parameters

    def \_\_init\_\_(self, cache\_manager: RAM\_cache | None = DB\_Cache(), cache\_allowed: bool = True, parallel=True, min\_parallel\_depth=2, workers=get\_cores(), \*\*kwargs) -> None:

        self.parallel = parallel

        if parallel and cache\_allowed:

            assert isinstance(cache\_manager, DB\_Cache), "Only DB cache an be used in parallel"

        self.min\_parallel\_depth = min\_parallel\_depth

        self.workers = workers

        # use parent class construction

        super().\_\_init\_\_(cache\_manager=cache\_manager, cache\_allowed=cache\_allowed, \*\*kwargs)

    def should\_use\_parallel(self, board\_state: Board\_State, depth):

        return depth >= self.min\_parallel\_depth and self.parallel

    # this function breaks up the legal moves array into many sub arrays that have a similar distribution of good and bad moves

    def break\_up\_legal\_moves\_to\_segments(self, legal\_moves: list, number\_sub\_arrays):

        # print("Legal moves:")

        # print(legal\_moves)

        legal\_move\_sub\_arrays = [[] for \_ in range(number\_sub\_arrays)]

        # repeatedly add next best move to the end of the sub array

        # iterating through the sub arrays to give each one a similar distribution of move quality

        i = 0

        while legal\_moves:

            legal\_move\_sub\_arrays[i].append(

                legal\_moves.pop(0)

            )

            i = (i+1) % number\_sub\_arrays

        # reverse sub arrays so in order of best to worst

        legal\_move\_sub\_arrays = list(map(

            lambda e: list(reversed(e)),

            legal\_move\_sub\_arrays

        ))

        # print("Broken down")

        # for sub\_array in legal\_move\_sub\_arrays:

        #     print(sub\_array)

        return legal\_move\_sub\_arrays

    # this function runs the presort moves process in parallel

    def generate\_move\_child\_in\_parallel(self, board\_state: Board\_State, depth: int, is\_maximizer, give\_child=True, max\_time=None):

        # print("generate\_move\_child\_in\_parallel called")

        # self interrupting

        if is\_time\_expired(max\_time):

            raise TimeOutError()

        # explore\_depth = max(depth-3, 0)

        explore\_depth = max(depth-2, 0)

        if (not self.parallel):

            yield from super().generate\_move\_child(

                board\_state=board\_state,

                depth=depth,

                is\_maximizer=is\_maximizer,

                give\_child=give\_child,

                # is\_time\_expired=is\_time\_expired

                max\_time=max\_time

            )

        else:

            # if can do presort concurrently

            #   # print("getting legal moves")

            legal\_moves = list(board\_state.generate\_legal\_moves())

            # print("setting up job")

            # create job

            job = Presort\_Moves\_Sub\_Job(

                depth=explore\_depth,

                board\_state=board\_state,

                cache\_allowed = self.cache\_allowed,

                max\_time=max\_time,

            )

            # concurrently map the job onto the iterable of legal moves

            # print("generate\_move\_child\_in\_parallel: Using multiprocessing pool to presort in parallel")

            with multiprocessing.Pool(self.workers) as pool:

                moves\_and\_scores = pool.map(

                    func=job,

                    iterable=legal\_moves

                )

            # print("generate\_move\_child\_in\_parallel: finished with multiprocessing pool")

            # self interrupting

            if is\_time\_expired(max\_time):

                raise TimeOutError()

            # sort by score

            moves\_and\_scores = sorted(

                moves\_and\_scores,

                key=lambda triplet: triplet[0],

                reverse=is\_maximizer

            )

            # use map to get rid of score

            def move\_and\_blank\_child(score\_and\_move):

                \_, move = score\_and\_move

                return (move, None)

            moves\_scores\_children = map(move\_and\_blank\_child, moves\_and\_scores)

            # print("children sorted by score")

            yield from moves\_scores\_children

    # this function performs a minimax search in parallel

    def parallel\_minimax(self, board\_state, depth, max\_time=None, \*args, \*\*kwargs):

        # print("Call  parallel\_minimax")

        # self interrupting

        if is\_time\_expired(max\_time):

            raise TimeOutError()

        # print("parallel\_minimax, checking cache")

        # if cache can be used then check if this search has already been complete

        if self.cache\_allowed:

            result = self.cache\_manager.search\_cache(

                board\_state=board\_state,

                depth=depth

            )

            if result is not None:

                # print("no call needed as cache is sufficient")

                return result["score"], result["move"]

        # print("Parallel minimax called")

        # assert isinstance(depth, int)

        # assert depth >= 2, "depth must be greater or equal to 2 to do parallelization"

        is\_maximizer = self.color\_maximizer\_key.get(board\_state.next\_to\_go)

        # print("started getting moves\_and\_child\_sorted")

        # print("getting legal moves, presorted")

        # generate presorted moves iterable

        # print("parallel\_minimax: calling generate\_move\_child\_in\_parallel")

        moves\_sorted = list(self.generate\_move\_child\_in\_parallel(

            board\_state=board\_state,

            depth=depth,

            is\_maximizer=is\_maximizer,

            give\_child=False,

            # is\_time\_expired=is\_time\_expired,

            max\_time=max\_time,

        ))

        # print("parallel\_minimax: finished generate\_move\_child\_in\_parallel")

        # print("finished getting moves\_and\_child\_sorted")

        # print("moves\_sorted: ")

        # print(moves\_sorted)

        # break up these pre sorted legal moves into segments for each sub job to work on

        # print("parallel\_minimax, breaking up moves into sub arrays")

        legal\_move\_sub\_arrays = self.break\_up\_legal\_moves\_to\_segments(

            legal\_moves=moves\_sorted,

            number\_sub\_arrays=self.workers

        )

        # print("parallel\_minimax: finished break\_up\_legal\_moves\_to\_segments")

        # self interrupting

        if is\_time\_expired(max\_time):

            raise TimeOutError()

        # print("legal moves sub arrays generated")

        # print("Printing legal moves sub arrays")

        # print(legal\_move\_sub\_arrays)

        # print("Defining job for workers")

        # print("parallel\_minimax, defining sub job")

        # use these legal moves to do many simultaneous minimax operations

        # I updated the sub job classes so that I don't need to pass a cache manager object to them

        # this was causing issues as the database connection that the database cache manager contained couldn't be piped between workers

        # # create a job function

        # job = Minimax\_Sub\_Job(

        #     # move\_engine=self,

        #     move\_engine=Move\_Engine(

        #         cache\_allowed=False,

        #         cache\_manager=None,

        #         # cache\_manager=cache\_manager,

        #         # cache\_allowed=cache\_allowed,

        #         # is this the miracle fix?

        #         additional\_depth=self.additional\_depth,

        #         use\_validator=self.use\_validator,

        #     ),

        #     board\_state=board\_state,

        #     depth=depth,

        #     # is\_time\_expired = is\_time\_expired,

        #     max\_time=max\_time,

        #     args=args,

        #     kwargs=kwargs,

        # )

        # create a job function

        job = Minimax\_Sub\_Job(

            board\_state=board\_state,

            depth=depth,

            max\_time=max\_time,

            additional\_depth=self.additional\_depth,

            cache\_allowed=self.cache\_allowed,

            args=args,

            kwargs=kwargs,

        )

        # print(f"Using pool to complete jobs in parallel (pool size = {cores})")

        # map this job function onto the legal moves sub array in parallel

        # print("parallel\_minimax: Using multiprocessing pool")

        with multiprocessing.Pool(self.workers) as pool:

            minimax\_sub\_job\_results = pool.map(

                func=job,

                iterable=legal\_move\_sub\_arrays

            )

        # self interrupting

        if is\_time\_expired(max\_time):

            raise TimeOutError()

        # print("parallel\_minimax: multiprocessing finished")

        # set a best move and score variable with starting values

        best\_move = None

        if is\_maximizer:

            best\_score = 0-(ARBITRARILY\_LARGE\_VALUE + 1)

        else:

            best\_score = ARBITRARILY\_LARGE\_VALUE + 1

        # select best move

        for result in minimax\_sub\_job\_results:

            score, move = result

            # if this move is better then update the best move and score variables

            if (is\_maximizer and score > best\_score) or (not is\_maximizer and score < best\_score):

                best\_score = score

                best\_move = move

        # add the result to the cache

        if self.cache\_allowed:

            self.cache\_manager.add\_to\_cache(

                board\_state=board\_state,

                depth=depth,

                move=best\_move,

                score=best\_score

            )

        # print("finished getting best outcome")

        return best\_score, best\_move

    # this function is able to perform handle the initial non recursive call for a parallel minimax search

    def minimax\_first\_call\_parallel(self, board\_state: Board\_State, depth, max\_time=None, \*args, \*\*kwargs):

        # print(f"CALL   minimax\_first\_call\_parallel")

        # print(f"CALL minimax\_first\_call\_parallel(self, board\_state={hash(board\_state)}, depth={depth}, \*args, \*\*kwargs)")

        # assert isinstance(depth, int)

        # self interrupting

        if is\_time\_expired(max\_time):

            raise TimeOutError()

        should\_use\_parallel = self.should\_use\_parallel(board\_state=board\_state, depth=depth)

        # print(f"minimax\_first\_call\_parallel:   should\_use\_parallel   -->   {should\_use\_parallel}")

        # self interrupting

        if is\_time\_expired(max\_time):

            raise TimeOutError()

        # I find the below code clearer, but unfortunately it won't work as the functions cannot be pickled and sent/piped between the workers

        # if should\_use\_parallel:

        #     def chosen\_minimax\_function():

        #         return self.parallel\_minimax(

        #             board\_state=board\_state,

        #             depth=depth,

        #             cache\_allowed=self.cache\_allowed,

        #             cache\_manager=self.cache\_manager,

        #             # is\_time\_expired = is\_time\_expired,

        #             max\_time=max\_time,

        #             \*args,

        #             \*\*kwargs

        #         )

        # else:

        #     def chosen\_minimax\_function():

        #         return self.minimax(

        #             board\_state=board\_state,

        #             depth=depth,

        #             max\_time=max\_time,

        #             \*args, \*\*kwargs

        #         )

        # if self.cache\_allowed:

        #     with self.cache\_manager:

        #         result = chosen\_minimax\_function()

        # else:

        #     result = chosen\_minimax\_function()

        # the below code simply make 2 decisions,

        # should parallel minimax be used or vanilla minimax?

        # should cache be used?

        # it is unfortunate that the above function based solution didn't work as be bellow code features lots of repetition

        if should\_use\_parallel:

            if self.cache\_allowed:

                with self.cache\_manager:

                    result = self.parallel\_minimax(

                        board\_state=board\_state,

                        depth=depth,

                        max\_time=max\_time,

                        \*args,

                        \*\*kwargs

                    )

            else:

                result = self.parallel\_minimax(

                    board\_state=board\_state,

                    depth=depth,

                    max\_time=max\_time,

                    \*args,

                    \*\*kwargs

                )

        else:

            if self.cache\_allowed:

                with self.cache\_manager:

                    result = self.minimax(

                        board\_state=board\_state,

                        depth=depth,

                        max\_time=max\_time,

                        \*args, \*\*kwargs

                    )

            else:

                result = self.minimax(

                    board\_state=board\_state,

                    depth=depth,

                    max\_time=max\_time,

                    \*args, \*\*kwargs

                )

        if self.use\_validator:

            self.validator(result=result, board\_state=board\_state)

        return result

    # when the object is called, use minimax\_first\_call\_parallel to handle the call

    def \_\_call\_\_(self, \*args, \*\*kwargs):

        return self.minimax\_first\_call\_parallel(\*args, \*\*kwargs)

# this move engine class represents be best configuration of the move engine

# it also updates some functions form parallel minimax (improves upon it)

class Move\_Engine\_Prime(Parallel\_Move\_Engine):

    def \_\_init\_\_(self) -> None:

        super().\_\_init\_\_(

            parallel=True,

            cache\_allowed=True,

            cache\_manager=DB\_Cache(min\_DB\_depth=1),

            additional\_depth=1,

            # additional\_depth=0,

            presort\_moves=True,

            color\_maximizer\_key={"W": True, "B": False},

            use\_validator=False,

            # workers = get\_cores()

            workers=5

        )

        self.depth = 2

    # when called, use standard depth if no depth parameter provided

    def \_\_call\_\_(self, board\_state: Board\_State, depth=None, \*args, \*\*kwargs):

        if depth is None:

            return super().\_\_call\_\_(board\_state=board\_state, depth=self.depth, \*args, \*\*kwargs)

        else:

            return super().\_\_call\_\_(board\_state=board\_state, depth=depth, \*args, \*\*kwargs)

    # this function returns a boolean to decide if parallel processing is needed.

    # it takes the board state as a parameter as the evaluation could depend on the legal moves to analyze

    def should\_use\_parallel(self, board\_state: Board\_State, depth):

        # if not allowed to use parallel or hte game is over then return false

        if not self.parallel:

            return False

        over, \_ = board\_state.is\_game\_over\_for\_next\_to\_go()

        if over:

            return False

        # otherwise begin by estimating the depth that the decision tree will be searched to

        if board\_state.check\_encountered:

            likely\_depth = depth + 0.5 \* self.additional\_depth

        else:

            likely\_depth = depth

        # then estimate the branching factor by looking at the average legal moves available to both players

        board\_state\_a = board\_state

        legal\_moves\_a = list(board\_state.generate\_legal\_moves())

        some\_legal\_move\_a = legal\_moves\_a[0]

        board\_state\_b = board\_state\_a.make\_move(\*some\_legal\_move\_a)

        branching\_factor\_a = len(legal\_moves\_a)

        branching\_factor\_b = len(list(board\_state\_b.generate\_legal\_moves()))

        likely\_branching\_factor = 1/2 \* (branching\_factor\_a + branching\_factor\_b)

        # use an exponential formula to create and estimate for static evaluations

        estimated\_static\_eval = likely\_branching\_factor \*\* likely\_depth

        # print({"estimated\_static\_eval": estimated\_static\_eval})

        # if the number of evaluations exceeds that of depth 2 on the starting positions, then use parallel

        return (estimated\_static\_eval >= 400)

    def break\_up\_legal\_moves\_to\_segments(self, legal\_moves: list, number\_sub\_arrays):

        # use the original break up legal moves function as it is better

        return super().break\_up\_legal\_moves\_to\_segments(legal\_moves, number\_sub\_arrays)

        # the bellow code was experimental and turned out to not improve performance,

        # it divided the scored legal move up differently into sub arrays

        # it put the best few in the first sub array and then the next best few in the next and so on

        # this was worse than the previous function. this is because creating a distribution or good and bad from best to worst improved the pruning

        # legal\_move\_sub\_arrays = [[] for \_ in range(number\_sub\_arrays)]

        # number\_legal\_moves = len(legal\_moves)

        # number\_workers = self.workers

        # moves\_per\_worker = number\_legal\_moves // number\_workers

        # workers\_with\_extra = number\_legal\_moves % number\_workers

        # workers\_without\_extra = number\_workers - workers\_with\_extra

        # moves\_per\_sub\_array = [moves\_per\_worker] \* workers\_without\_extra + [moves\_per\_worker+1] \* workers\_with\_extra

        # assert sum(moves\_per\_sub\_array) == number\_legal\_moves

        # # print(moves\_per\_sub\_array)

        # # >>> x = [1,2,3,4,5]

        # # >>> x[:2]

        # # [1, 2]

        # # >>> x[2:]

        # # [3, 4, 5]

        # for worker\_index in range(number\_workers):

        #     moves\_in\_array = moves\_per\_sub\_array[worker\_index]

        #     legal\_move\_sub\_arrays[worker\_index] = legal\_moves[:moves\_in\_array]

        #     legal\_moves = legal\_moves[moves\_in\_array:]

        # # print("Broken down")

        # # for sub\_array in legal\_move\_sub\_arrays:

        # #     print(sub\_array)

        # yield from legal\_move\_sub\_arrays

# this minimax algorithm explores the decision tree to a given depth before stopping

class Move\_Engine\_Timed(Move\_Engine\_Prime):

    # explore for a certain amount of time

    def timed\_call(self, board\_state: Board\_State, time):

        # so that internal jobs can access the time used up indicator

        # self.max\_time = perf\_counter() + time

        # take 3 seconds to close threads

        time\_delta\_allowed = time

        # time\_delta\_allowed = max(time - 3, 0)

        # time\_delta\_allowed = max(time - 10, 0)

        # time\_delta\_allowed = max(time - 4, 0.5)

        # print({"time\_delta\_allowed": time\_delta\_allowed})

        # stop at max time

        self.max\_time = perf\_counter() + time\_delta\_allowed

        def time\_used\_up():

            return perf\_counter() >= self.max\_time

        # need minimum of depth 1 as depth 0 doesn't give a move

        # ensure that even if time = 0, at least a depth 1 search is completed

        # deepest\_result = self.minimax\_first\_call(board\_state=board\_state, depth=1)

        deepest\_result = self.minimax\_first\_call(board\_state=board\_state, depth=1, variable\_depth=0)

        depth = 2

        def absolute(x): return x if x >= 0 else 0-x

        # while True:

        # while there is time left, keep searching to a greater depth

        while not time\_used\_up():

            # print(f"Iterating again: time\_used\_up()   -->   {time\_used\_up()}")

            try:

                # deepest\_result = self.minimax\_first\_call\_parallel(board\_state=board\_state, depth=depth, is\_time\_expired=time\_used\_up)

                # search the tree in parallel if needed

                result = self.minimax\_first\_call\_parallel(board\_state=board\_state, depth=depth, max\_time=self.max\_time, variable\_depth=0)

                score, move = result

                # if the move or score are invalid then raise the appropriate error

                if absolute(score) == 1\_000\_001 or move is None:

                    raise UnexplainedErroneousMinimaxResultError()

            except UnexplainedErroneousMinimaxResultError:

                # I don't know the cause of the error but I can catch it here and prevent it causing issues and producing unexpected behaviour

                break

            except TimeOutError:

                # print(f"Breaking: time\_used\_up()   -->   {time\_used\_up()}")

                break

            else:

                # print(f"Completes depth={depth} so incrementing depth")

                # print(f"Result at depth {depth}:   {deepest\_result}")

                # if neither the time is used up or an erroneous result was produced, keep searching at a greater depth

                deepest\_result = result

                depth += 1

        # return the best move

        move, \_ = deepest\_result

        assert move is not None

        # print(f"Returning result at greatest depth={depth}")

        return deepest\_result, depth

    def \_\_call\_\_(self, board\_state: Board\_State, time, \*args, \*\*kwargs):

        # use the timed\_call handler function

        assert not board\_state.is\_game\_over\_for\_next\_to\_go()[0]

        # start = perf\_counter()

        deepest\_result, \_ = self.timed\_call(board\_state=board\_state, time=time)

        # end = perf\_counter()

        # time\_delta = end - start

        # print(f"timed minimax set to {time} seconds actually took {round(time\_delta, 3)} seconds")

        return deepest\_result

# the below function was used to take benchmarks of the timed minimax algorithm in order to gauge how accurately it stuck to its time limit

def check\_timed():

    board\_state = Board\_State()

    move\_engine = Move\_Engine\_Timed()

    time = 0

    # for \_ in range(6):

    #     start = perf\_counter()

    #     result, depth = move\_engine(board\_state, 1)

    #     time += perf\_counter() - start

    #     print(f"At total time {time} sec: depth={depth};  result={result}")

    # for \_ in range(6):

    #     start = perf\_counter()

    #     result, depth = move\_engine(board\_state, 2)

    #     time += perf\_counter() - start

    #     print(f"At total time {time} sec: depth={depth};  result={result}")

    # for \_ in range(6):

    #     start = perf\_counter()

    #     result, depth = move\_engine(board\_state, 5)

    #     time += perf\_counter() - start

    #     print(f"At total time {time} sec: depth={depth};  result={result}")

    # for \_ in range(6):

    #     start = perf\_counter()

    #     result, depth = move\_engine(board\_state, 10)

    #     time += perf\_counter() - start

    #     print(f"At total time {time} sec: depth={depth};  result={result}")

    # for \_ in range(6):

    #     start = perf\_counter()

    #     result, depth = move\_engine(board\_state, 15)

    #     time += perf\_counter() - start

    #     print(f"At total time {time} sec: depth={depth};  result={result}")

    while True:

        start = perf\_counter()

        result, depth = move\_engine(board\_state, 20)

        time += perf\_counter() - start

        print(f"At total time {time} sec: depth={depth};  result={result}")

if \_\_name\_\_ == "\_\_main\_\_":

    check\_timed()

The above code contains 3 main classes and 3 sub classes.

Parallel\_Move\_Engine inherits from the vanilla move engine and adds concurrency

Move\_Engine\_Prime inherits from Parallel\_Move\_Engine and adds the optimal configurations by default. It also improves on some of the methods.

Move\_Engine\_Timed inherits from Move\_Engine\_Prime and includes the ability to search the minimax tree for a given amount of time before stopping and using an insurance search.

Parallel\_Move\_Engine uses the multiprocessing library to make use of all the cores on my computer when performing a computationally intensive minimax search.

It parallelises 2 parts of the vanilla minimax algorithm. It firstly evaluates the child nodes for pre-sorting in parallel. Then if breaks up the array of child nodes into sub arrays (breaks the tree into sub trees). It then searches each of these sub arrays of child nodes (sub trees) in parallel. This greatly improved time efficiency as the minimax function was limited by my processor speed.

To achieve this, the main obstacle was ensuring that all my object were pickleable. This allowed them to be sent to each individual worker. Because of this, I had to use callable objects rather than functions. This is why I created Presort\_Moves\_Sub\_Job and Minimax\_Sub\_Job. Each of these represents a function that will be executed many times on many different parameters concurrently by different workers running on different cores of my CPU.

Minimax prime improved the method that decided whether or not to use the parallel minimax function. I decide to use 400+ static evaluations as the cut of point as I found that evaluating the starting positions at depth 2 took a similar amount of time in parallel as using the vanilla minimax algorithm (10-12 seconds).

I decided to use 5 workers as my laptops CPU has 4 physical cores and 8 logical cores. I found that this meant that between 4 and 8 task could be completed concurrently. I realised that when I used 4 workers, not all the CPU’s power was used. As a result I decided to use 5 workers. I wanted to keep the number of workers as low as possible while fully using the whole CPU. This was in an effort to maximise the size of the legal move sub arrays that each worker was processing. This allowed for increased opportunities for pruning.

Graphical user interface, application, table

Description automatically generated

Here you can see that a parallel minimax search of depth 2 of the starting positions has created 5 independent python instances in the task manager. These instances are using almost all the CPU’s power and are each performing a Minimax\_Sub\_Job function on a different segment of the legal moves.

The Move\_Engine\_Timed was my final iteration of the minimax algorithm. It features all the optimisations that I planned in my design section. This means that it is efficient and controllable as I can now more easily limit how much it can explore the decision tree. This allow me to make more difficulty setting and to make the algorithm more consistent and predictable in how long it takes.

Game Manager:

I added to the main Game class from prototype 2. The main edition was code to more easily represent a console game of chess. This was used in the console chess program as well as in unit tests.

# import other modules

from chess\_functions import Board\_State, Vector

from move\_engine import Move\_Engine\_Timed

from assorted import NotUserTurn, NotComputerTurn, InvalidMove, safe\_hash

from time import perf\_counter

import os

# this function wipes the console to allow the updated board to be displayed

def clear\_console():

    #  https://stackoverflow.com/questions/517970/how-to-clear-the-interpreter-console

    os.system('cls')

# this function allows a function's execution time to be measured

def time\_function(function):

    start = perf\_counter()

    result = function()

    end = perf\_counter()

    time\_delta = end - start

    return result, time\_delta

# the game class is used to keep track of a chess game between a user and the computer

class Game(object):

    # constructor for game object

    def \_\_init\_\_(self, time=10, user\_color="W", echo=False) -> None:

        # based on user's color, determine color key

        self.player\_color\_key = {

            "W": 1 if user\_color=="W" else -1,

            "B": -1 if user\_color=="W" else 1

        }

        # set depth property from parameters

        self.time = time

        self.echo = echo

        # set attributes for game at start

        self.board\_state = Board\_State()

        self.move\_counter = 0

        self.move\_engine = Move\_Engine\_Timed()

        self.game\_history\_output = ()

    # hash function describes all unique properties of the game as a unique number

    def \_\_hash\_\_(self):

        return hash(safe\_hash((

            "Game",

            self.player\_color\_key,

            self.time,

            self.echo,

            self.board\_state,

            self.move\_counter,

            self.move\_engine,

            self.game\_history\_output,

        )))

    # thi function determines if 2 game objects are equal

    def \_\_eq\_\_(self, other: object) -> bool:

        if not isinstance(other, Game):

            return False

        return hash(other) == hash(other)

    @staticmethod

    def create\_row(moving\_player, time\_delta, move\_count, new\_utility, future\_utility, move\_description, white\_pieces\_taken, black\_pieces\_taken, number\_legal\_moves) -> str:

        # print("(moving\_player, time\_delta, move\_count, new\_utility, future\_utility, move\_description, white\_pieces\_taken, black\_pieces\_taken, number\_legal\_moves)")

        # print((moving\_player, time\_delta, move\_count, new\_utility, future\_utility, move\_description, white\_pieces\_taken, black\_pieces\_taken, number\_legal\_moves))

        # this function creates formatted string that represents a row in a table and contains all the data provided as parameters

        if time\_delta is None:

            time\_delta = ""

        if future\_utility is None:

            future\_utility = ""

        return f"| {move\_count:^14} | {moving\_player:^15} | {new\_utility:^15}  | {future\_utility:^15}  | {time\_delta:^15} | {number\_legal\_moves:^15} | {move\_description:<40} | {white\_pieces\_taken:<35} | {black\_pieces\_taken:<35} |"

    def print\_game\_history(self):

        # this function clears the console and then print out a table that contains the history of the game

        clear\_console()

        # https://www.geeksforgeeks.org/string-alignment-in-python-f-string/

        print("Moves History:")

        print()

        # print table header

        # print(self.create\_row('Moving Color', 'Move Count', 'Time Taken (sec)', 'New Utility', 'Move', 'White Pieces Taken', 'Black Pieces Taken'))

        print(

            self.create\_row(

                moving\_player="Moving Color",

                time\_delta="Time Taken",

                new\_utility="New Utility",

                future\_utility="Future Utility",

                move\_description="Move Description",

                white\_pieces\_taken="White Pieces Taken",

                black\_pieces\_taken="Black Pieces Taken",

                move\_count="Move Number",

                number\_legal\_moves="NO legal moves"

            )

        )

        # print each row

        for item in self.game\_history\_output:

            print(item)

        print()

        # print the board

        self.board\_state.print\_board()

    def make\_move(self, move, time\_delta, future\_utility):

        # if echo, gather some data before the move

        if self.echo:

            color\_moving = "White (+)" if self.board\_state.next\_to\_go == "W" else "Black (-)"

            # print({"move": move})

            position\_vector: Vector = move[0]

            resultant\_vector: Vector = move[0] + move[1]

            moving\_piece, taken\_piece = self.board\_state.get\_piece\_at\_vector(position\_vector), self.board\_state.get\_piece\_at\_vector(resultant\_vector)

            from\_square, to\_square = position\_vector.to\_square(), resultant\_vector.to\_square()

            no\_legal\_moves = self.board\_state.number\_legal\_moves

        # adjust properties that keep track of the game state

        # implement the move on the board state

        self.board\_state = self.board\_state.make\_move(\*move)

        # print(f"self.board\_state.next\_to\_go   -->   {self.board\_state.next\_to\_go}")

        self.move\_counter += 1

        # if echo, gather move data, format and create a row and then print out the move history

        if self.echo:

            # find various data points

            utility\_score = self.board\_state.static\_evaluation()

            move\_number = self.move\_counter

            if taken\_piece is None:

                move\_description = f"{moving\_piece} moved from {from\_square} to {to\_square}"

            else:

                move\_description = f"{moving\_piece} moved from {from\_square} to {to\_square}, taking piece {taken\_piece}"

            def pieces\_missing\_characters(color):

                return list(map(

                    str,

                    self.board\_state.generate\_pieces\_taken\_by\_color(color)

                ))

            white\_pieces\_taken = " ".join(pieces\_missing\_characters("W"))

            # white\_pieces\_taken = "(No pieces taken)" if white\_pieces\_taken == "" else white\_pieces\_taken

            black\_pieces\_taken = " ".join(pieces\_missing\_characters("B"))

            # black\_pieces\_taken = "(No pieces taken)" if black\_pieces\_taken == "" else black\_pieces\_taken

            if time\_delta is not None:

                time\_delta = f"{round(time\_delta, 2)} sec"

            # else:

            #     time\_delta = ""

            # create a row

            new\_row = self.create\_row(

                moving\_player=color\_moving,

                time\_delta=time\_delta,

                move\_count=move\_number,

                new\_utility=utility\_score,

                move\_description=move\_description,

                white\_pieces\_taken=white\_pieces\_taken,

                black\_pieces\_taken=black\_pieces\_taken,

                number\_legal\_moves=no\_legal\_moves,

                future\_utility=future\_utility,

            )

            # add the row to the history

            self.game\_history\_output = tuple(list(self.game\_history\_output) + [new\_row])

            # create a now row to show check if appropriate

            if self.board\_state.color\_in\_check():

                check\_msg = f"CHECK:  {self.board\_state.next\_to\_go} in check"

                new\_row = self.create\_row(

                    moving\_player="-",

                    time\_delta="-",

                    move\_count="-",

                    new\_utility="-",

                    move\_description=check\_msg,

                    white\_pieces\_taken="-",

                    black\_pieces\_taken="-",

                    number\_legal\_moves="-",

                    future\_utility="-",

                )

                self.game\_history\_output = tuple(list(self.game\_history\_output) + [new\_row])

            # create a new row to show game over if appropriate

            over, winner = self.board\_state.is\_game\_over\_for\_next\_to\_go()

            if over:

                how = "Stalemate" if winner is None else "Checkmate"

                winning\_color = "White" if winner == "W" else "B"

                if how == "Stalemate":

                    if self.board\_state.is\_3\_board\_repeats\_in\_game\_history():

                        game\_over\_msg = "Stalemate: both players draw (3 repeats of the board state)"

                    else:

                        game\_over\_msg = "Stalemate: both players draw (no legal moves and not in check)"

                else:

                    game\_over\_msg = f"Checkmate: {winning\_color} wins"

                new\_row = self.create\_row(

                    moving\_player="-",

                    time\_delta="-",

                    move\_count="-",

                    new\_utility="-",

                    move\_description=game\_over\_msg,

                    white\_pieces\_taken="-",

                    black\_pieces\_taken="-",

                    number\_legal\_moves="-",

                    future\_utility="-",

                )

                self.game\_history\_output = tuple(list(self.game\_history\_output) + [new\_row])

            # print the game history

            self.print\_game\_history()

    # this function validates if the user's move is allowed and if so, makes it

    def implement\_user\_move(self, from\_square, to\_square, time\_delta=None, estimated\_utility=None) -> None:

        # check that the user is allowed to move

        which\_player\_next\_to\_go = self.player\_color\_key.get(

            self.board\_state.next\_to\_go

        )

        if which\_player\_next\_to\_go != 1:

            raise NotUserTurn(game=self)

        # unpack move into vector form

        # invalid square syntaxes will cause a value error here

        try:

            position\_vector = Vector.construct\_from\_square(from\_square)

            movement\_vector = Vector.construct\_from\_square(to\_square) - position\_vector

        except Exception:

            raise ValueError("Square's not in valid format")

        move = (position\_vector, movement\_vector)

        # if the move is not in the set of legal moves, raise and appropriate exception

        if move not in self.board\_state.generate\_legal\_moves():

            raise InvalidMove(game=self)

        self.make\_move(move, time\_delta, estimated\_utility)

        return move, self.board\_state.static\_evaluation()

    # this function determines if the game is over and if so, what is the nature of the outcome

    def check\_game\_over(self) -> list[bool, str, str]:

        # returns: over: bool, winning\_player: (1/-1), classification: str

        # determine if board state is over for next player

        over, winner = self.board\_state.is\_game\_over\_for\_next\_to\_go()

        # switch case statement to determine the appropriate values to be returned in each case

        match (over, winner):

            case False, \_:

                victory\_classification = None

                winning\_player = None

            case True, None:

                if self.board\_state.is\_3\_board\_repeats\_in\_game\_history():

                    victory\_classification = "stalemate board repeat"

                else:

                    victory\_classification = "stalemate no legal moves"

                winning\_player = None

            case True, winner:

                victory\_classification = "checkmate"

                winning\_player = self.player\_color\_key[winner]

        # return appropriate values

        return over, winning\_player, victory\_classification

    # this function determines and implements the computer move

    def implement\_computer\_move(self, best\_move\_function=None):

        # for use with testing bots, a best move function can be provided for use, but minimax if the default

        # get next to go player (1/-1)

        which\_player\_next\_to\_go = self.player\_color\_key.get(

            self.board\_state.next\_to\_go

        )

        # check that is it the computer's turn

        if which\_player\_next\_to\_go != -1:

            raise NotComputerTurn()

            # raise ValueError(f"Next to go is user: {self.board\_state.next\_to\_go} not computer")

        # if no function provided, default to minimax

        if best\_move\_function is None:

            def best\_move\_function(self):

                return self.move\_engine(

                    board\_state = self.board\_state,

                    # depth is based of difficulty of game based on depth parameter

                    time = self.time,

                )

        # otherwise use provided function,

        # the provided function should take game as an argument and then return data in the same format as the minimax function

        result, time\_delta = time\_function(

            lambda: best\_move\_function(self)

        )

        # print({"result": result})

        score, best\_move = result

        # best\_move, score = result

        # print({"best\_move": best\_move})

        # print({"score": score})

        assert best\_move is not None

        self.make\_move(best\_move, time\_delta, score)

        # incase is it wanted for a print out ect, return move and score

        return best\_move, score

Below is an example of the additional functionality being used to show what is happening in a unit test of depth 3 bot playing a depth 2 bot

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A computer screen capture

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

I then created a new class called Game\_Website. It is responsible to keep track of a chess game that is going in on the client side on my webpage user interface. It originally inherited from the Game class but then I began overwriting so many methods that it was complicating the class. As a result the Game\_Website class is similar but separate from the game class.

Game\_web.py

# this class is a variant of the game class that is to be used in a web game of chess

# it originally inherited from the game class but then it became to different so I no longer made it inherit from the game class

# from .game import Game

from assorted import safe\_hash

from chess\_functions import Board\_State, Vector

from move\_engine import Move\_Engine\_Timed

from schemas import serialize\_piece, serialize\_move

# import other modules

from chess\_functions import Board\_State, Vector

from move\_engine import Move\_Engine\_Timed

from assorted import NotUserTurn, NotComputerTurn, InvalidMove, safe\_hash

from time import perf\_counter

import os

# # this function clears the console

# def clear\_console():

#     #  https://stackoverflow.com/questions/517970/how-to-clear-the-interpreter-console

#     os.system('cls')

# # this function allows for a functions performance to be measured

# def time\_function(function):

#     start = perf\_counter()

#     result = function()

#     end = perf\_counter()

#     time\_delta = end - start

#     return result, time\_delta

# the game class is used to keep track of a chess game between a user and the computer

class Game\_Website(object):

    # # it keeps track of:

    # # the player's color's

    # player\_color\_key: dict

    # # the difficulty or depth of the game

    # depth: int

    # # the current board state

    # board\_state: Board\_State

    # # the number of moves so far

    # move\_counter: int

    # move\_engine: Move\_Engine

    # game\_history\_output: tuple[str]

    # constructor for game object

    # def \_\_init\_\_(self, time, user\_color="W", echo=True) -> None:

    def \_\_init\_\_(self, difficulty="medium", user\_color="W") -> None:

        self.difficulty = difficulty.strip().lower()

        # based on user's color, determine color key

        self.player\_color\_key = {

            "W": 1 if user\_color == "W" else -1,

            "B": -1 if user\_color == "W" else 1

        }

        # set depth property from parameters

        # set attributes for game at start

        # start with a blank board

        self.board\_state = Board\_State()

        self.move\_counter = 0

        # use the timed move engine to find moves

        self.move\_engine = Move\_Engine\_Timed()

        self.move\_history\_output = []

    # the time function is treated as a property to emulate how it was before when it was a property (before difficulty)

    @property

    def time(self):

        # return 0.5

        match self.difficulty:

            case "really\_easy": return 1

            case "easy": return 2

            case "medium": return 5

            case "hard": return 10

            case "really\_hard": return 15

            case "extreme": return 30

            case "legendary": return 60

        raise ValueError(f"Difficulty {self.difficulty} not recognised")

    # this functions adds a move to the move history,

    # it doesn't make the move

    def add\_move\_to\_history(self, move):

        # should be called pre move

        # color\_moving = "White" if self.board\_state.next\_to\_go == "W" else "Black"

        # print({"move": move})

        # unpack move and determine squares and relevant pieces

        position\_vector: Vector = move[0]

        resultant\_vector: Vector = move[0] + move[1]

        moving\_piece, taken\_piece = self.board\_state.get\_piece\_at\_vector(position\_vector), self.board\_state.get\_piece\_at\_vector(resultant\_vector)

        from\_square, to\_square = position\_vector.to\_square(), resultant\_vector.to\_square()

        # increment move count

        self.move\_counter += 1

        # generate string move description using this data

        if taken\_piece is None:

            # move\_description = f"Move {self.move\_counter}: {color\_moving} moved {moving\_piece} from {from\_square} to {to\_square}"

            # move\_description = f"{self.move\_counter}: {color\_moving} moved {moving\_piece} from {from\_square} to {to\_square}"

            move\_description = f"{self.move\_counter}: {moving\_piece} from {from\_square} to {to\_square}"

        else:

            # move\_description = f"Move {self.move\_counter}: {color\_moving} moved {moving\_piece} from {from\_square} to {to\_square}, taking piece {taken\_piece}"

            # move\_description = f"{self.move\_counter}: {color\_moving} moved {moving\_piece} from {from\_square} to {to\_square}, taking piece {taken\_piece}"

            move\_description = f"{self.move\_counter}: {moving\_piece} from {from\_square} to {to\_square}, taking piece {taken\_piece}"

        # append this string to the move history list

        self.move\_history\_output.append(move\_description)

    # this function makes a move on the board state and logs this in the game history

    def make\_move(self, move):

        assert not self.board\_state.is\_game\_over\_for\_next\_to\_go()[0], "Cannot make move if game is over"

        assert move is not None, "Cannot move as move parameter not valid"

        self.add\_move\_to\_history(move)

        # adjust properties that keep track of the game state

        self.board\_state = self.board\_state.make\_move(\*move)

    # this function validates if the user's move is allowed and if so, makes it

    # input form javascript is a move in terms of vectors

    def implement\_user\_move(self, move: list[Vector]) -> None:

        assert not self.board\_state.is\_game\_over\_for\_next\_to\_go()[0], "Cannot make move if game is over"

        # check that the user is allowed to move

        which\_player\_next\_to\_go = self.player\_color\_key.get(

            self.board\_state.next\_to\_go

        )

        # check it is the user's go

        if which\_player\_next\_to\_go != 1:

            raise NotUserTurn(game=self)

        # if the move is not in the set of legal moves, raise and appropriate exception

        if move not in self.board\_state.generate\_legal\_moves():

            self.board\_state.print\_board()

            print(f"Move  {move} not in legal moves")

            raise InvalidMove(game=self)

        # make the move

        self.make\_move(move)

        # no return data needed

        # return move, self.board\_state.static\_evaluation()

    # this function determines if the game is over and if so, what is the nature of the outcome

    def check\_game\_over(self):

        # returns: over: bool, winning\_player: (W / B), classification: str

        # determine if board state is over for next player

        over, winner = self.board\_state.is\_game\_over\_for\_next\_to\_go()

        # switch case statement to determine the appropriate values to be returned in each case

        match (over, winner):

            case False, \_:

                victory\_classification = None

                winning\_player = None

            case True, None:

                if self.board\_state.is\_3\_board\_repeats\_in\_game\_history():

                    victory\_classification = "stalemate board repeat"

                else:

                    victory\_classification = "stalemate no legal moves"

                winning\_player = None

            case True, winner:

                victory\_classification = "checkmate"

                winning\_player = self.player\_color\_key[winner]

        # return appropriate values

        return over, winning\_player, victory\_classification

    # this function is used to determine the AI's move

    # the time delta can be explicitly given or it can use the time provided manually as a parameter

    def best\_move\_function(self, time):

        if time is None:

            time = self.time

        # returns just the move

        return self.move\_engine(

            board\_state=self.board\_state,

            # depth is based of difficulty of game based on depth parameter

            time=time,

        )[1]

    # this function is used to generate a description of the move that the computer has completed for the client side GUI

    # it is needed to provide highlighting ect.

    # it doesn't make the computer move

    def generate\_computer\_move\_description(self, move, previous\_board\_state):

        # determine various qualities about the move

        position\_vector, movement\_vector = move

        resultant\_vector = position\_vector + movement\_vector

        moved\_piece = previous\_board\_state.get\_piece\_at\_vector(position\_vector)

        taken\_piece = previous\_board\_state.get\_piece\_at\_vector(resultant\_vector)

        from\_square = position\_vector.to\_square()

        to\_square = resultant\_vector.to\_square()

        # return dictionary of this data ready for json serialization

        # to do this, ensure all objects such as pieces and vectors are serialized.

        move\_description = {

            "moved\_piece": serialize\_piece(moved\_piece),

            "taken\_piece": serialize\_piece(taken\_piece),

            "from\_square": from\_square,

            "to\_square": to\_square,

            "move": serialize\_move(move)

        }

        return move\_description

    # this function completed the computer's move in a given time delta and returns a description

    def implement\_computer\_move\_and\_report(self, time=None):

        previous\_board\_state = self.board\_state

        # get next to go player (1/-1)

        which\_player\_next\_to\_go = self.player\_color\_key.get(

            self.board\_state.next\_to\_go

        )

        # check that is it the computer's turn

        if which\_player\_next\_to\_go != -1:

            raise NotComputerTurn()

            # raise ValueError(f"Next to go is user: {self.board\_state.next\_to\_go} not computer")

        # get the computer's move

        best\_move = self.best\_move\_function(time=time)

        # defensive design to detect issues, ensure a move has been calculated

        assert best\_move is not None

        # implement the move and return a description

        self.make\_move(best\_move)

        return self.generate\_computer\_move\_description(best\_move, previous\_board\_state)

    def \_\_eq\_\_(self, other: object) -> bool:

        # this method used the hash function to determine if 2 of these objects are the same

        if not isinstance(other, Game\_Website):

            return False

        return hash(self) == hash(other)

    def \_\_hash\_\_(self):

        # this function encodes all the data that makes a game unique in a hash to be stored in a database

        return hash(safe\_hash((

            "Game\_Website",

            self.player\_color\_key,

            self.time,

            self.board\_state,

            self.move\_counter,

            # self.move\_engine,

            self.move\_history\_output,

            self.difficulty

        )))

The main changes are:

* It has no functionality to log the game to the console.
* It also produces a different game history array designed for the client side JavaScript to use
* It produces a description of the computer’s move to allow the client side JavaScript to display the move properly with highlighting. This description is in the form of a dictionary that includes pieces and vectors involved in the move.
* It uses the timed minimax algorithm to produce the computer move
* It also features different difficulty setting which correspond to a different exploration time by the timed minimax algorithm.
* The resulting object can also be hashed and pickled to binary form in order to be stored in a database.

The chess game (Game Manager) module export the following objects through its \_\_init\_\_.py file:

from .game import Game

# from .game\_with\_difficulty import Game\_With\_Difficulty

from .game\_web import Game\_Website

It exports the Game class and the Game\_Website class.

Database module:

The database module is responsible for setting up and managing the database that the save game feature and minimax cache use.

The \_\_init\_\_.py file for the database module is unusual as it includes some logic:

# this file does 2 things:

# it decides which objects should be exported as part of the database module

# it initialise the database (creates a connection and creates all tables)

# objects to export

from .create\_database import create\_session, end\_session, create\_engines, end\_engines

from .models import Minimax\_Cache\_Item, create\_tables

from .handle\_games import get\_saved\_game, save\_game

# initialise database

engines\_dict = create\_engines(echo=False)

create\_tables(engines\_dict)

persistent\_DB\_engine = engines\_dict["persistent\_DB\_engine"]

volatile\_RAM\_engine = engines\_dict["volatile\_RAM\_engine"]

Its main responsibilities are to:

* Export the relevant functions and classes to allow the rest of the program to access the database
* Create the database connection and all the tables that should exist (if they don’t already).

Create\_database.py

# external modules used

import os

import sqlalchemy as sqla

from sqlalchemy.orm import scoped\_session, sessionmaker

from sqlalchemy.pool import QueuePool

# path to database file

DB\_path = os.path.join(os.getcwd(), 'database', 'database.db')

# this function creates 2 database engines, one in RAM and one is secondary storage

def create\_engines(echo):

    # echo means, should all the queries be printed out

    # caution with below code, could cause unexpected behavior if not thread safe, further research needed

    # I don't know much about threading in python, I find it confusing due to complexities around the global interpreter lock

    # as a result I had to just try various arguments to create these engines in a way that allowed me to use them with my flask server

    persistent\_DB\_engine = sqla.create\_engine(

        'sqlite:///' + DB\_path,

        echo=echo,

        poolclass=QueuePool,

        connect\_args={'check\_same\_thread': False}

    )

    volatile\_RAM\_engine = sqla.create\_engine(

        "sqlite:///:memory:",

        echo=echo,

        poolclass=QueuePool,

        connect\_args={'check\_same\_thread': False}

    )

    return {

        "persistent\_DB\_engine": persistent\_DB\_engine,

        "volatile\_RAM\_engine": volatile\_RAM\_engine

    }

# this function safely terminates the database connection

def end\_engines(engines):

    for engine in engines.values():

        engine.dispose()

# this function creates a session to access the database

def create\_session(engine):

    return scoped\_session(sessionmaker(bind=engine))

# this function safely ends a session

def end\_session(session):

    session.commit()

    session.close()

this file contains the relevant functions to create and safely destroy both engine objects (the master connection to the data base) and session objects (a temporary connection used when actively querying that database).

It creates an engine corresponding to both a persistent database in a “database.db” file and a volatile database in RAM.

Models.py

# import external modules

import sqlalchemy as sqla

from sqlalchemy.ext.declarative import declarative\_base

import pickle

# cannot be used for type hings as it has a high risk of causing a circular import

# from chess\_game import Game\_Website

# create a base object with which I will make my database tables (part of the ORM)

Base = declarative\_base()

# create an object and a table (ORM used) to represent a cached minimax call

class Minimax\_Cache\_Item(Base):

    # here is the metadata and the columns of the database

    \_\_tablename\_\_ = "Minimax\_Cache"

    primary\_key = sqla.Column(sqla.Integer, primary\_key=True)

    board\_state\_hash = sqla.Column(sqla.String())

    depth = sqla.Column(sqla.INT())

    score = sqla.Column(sqla.INT())

    # 4 character move string encoded by the 4 digits of the from and movement vectors

    move = sqla.Column(sqla.String)

    # a single entry can be created by initializing an object with this class

    def \_\_init\_\_(self, board\_state\_hash: str, depth, move: str, score: int):

        # hash board state

        self.board\_state\_hash = str(board\_state\_hash)

        self.depth = depth

        self.move = move

        self.score = score

    # string description of object

    def \_\_repr\_\_(self) -> str:

        return f"Minimax\_Cache\_Item(board\_state\_hash='{self.board\_state\_hash}', depth={self.depth}, move='{self.move}', score={self.score})"

# this object represents an entry in a database table that saves games for later reloading

class Saved\_Game(Base):

    # metadata as well as columns defined here

    \_\_tablename\_\_ = "Saved\_Games"

    primary\_key = sqla.Column(sqla.Integer, primary\_key=True)

    # game\_hash = sqla.Column(sqla.String())

    cookie\_key = sqla.Column(sqla.String())

    raw\_game\_data = sqla.Column(sqla.BINARY())

    # initializing an object from this class allow a new entry in the database to be made

    def \_\_init\_\_(self, game, cookie\_key):

        self.cookie\_key = str(cookie\_key)

        self.raw\_game\_data = bytes(

            pickle.dumps(

                game

            )

        )

    # string description of object

    def \_\_repr\_\_(self) -> str:

        return f"Saved\_Game(cookie\_key='{self.cookie\_key}')"

# create indexes on cookie key and board state hash as these are the columns that I will use to search for a specific entry

sqla.schema.Index("board\_state\_hash", Minimax\_Cache\_Item.board\_state\_hash)

sqla.schema.Index("cookie\_key", Saved\_Game.cookie\_key)

# for each engine, build these tables

def create\_tables(engines\_dict):

    for engine in engines\_dict.values():

        Base.metadata.create\_all(engine, checkfirst=True)

This file contains a pair of classes that represent a single database entry within each of my 2 database table respectively. As the classes have been created using the SQL-Alchemy ORM, I can directly interact with the database by interacting with these objects.

After the tables are defined using the schema classes I then create and index for each table to set one of the columns to be a secondary key. I cannot create the tables without a standard incrementing primary key but I intend to use these secondary keys to query the tables.

The Saved\_Game table does store binary data as one of the columns. While in principle this can be inefficient, the binary data stored is very small (2-4 kb). This means it can be stored directly in the database without significant scalability issues.

The create tables function will create the relevant tables in each of the engines, once the engines are created.

Here is the handle\_games.py file:

# import external modules and local models

import pickle

from .models import Saved\_Game

# cannot import for type hint due to high risk of circular imports

# from chess\_game import Game\_Website

# this method queries the database for a game by cookie id

# and then deserializes the binary data to restore the game

def get\_saved\_game(cookie\_key, session):

    # query database for game by cookie key

    result: Saved\_Game = session.query(Saved\_Game)\

        .where(

            # Saved\_Game.game\_hash == str(hash(game\_hash))

            Saved\_Game.cookie\_key == str(cookie\_key)

        )\

            .first()

    # if the result is none (possible if browser has cookie that is invalid) then return none

    if result is None:

        # print(f"get\_saved\_game game not in database so returning none")

        return None

    # now deserialize the game from the binary data

    game = pickle.loads(

        result.raw\_game\_data

    )

    # print(f"this game recovered from database under cookie\_key:  {cookie\_key}")

    # game.board\_state.print\_board()

    # print(f"game recovered from database:  {hash(game)}")

    # return the game

    return game

# this function saves a game to the database under a certain cookie ID.

def save\_game(game, cookie\_key, session):

    # print("saving this game into database")

    # game.board\_state.print\_board()

    # print(f"saving game in database game with hash {hash(game)} under cookie\_key:  {cookie\_key}")

    # delete any old games with the same cookie id

    session.query(Saved\_Game)\

        .where(

            # Saved\_Game.game\_hash == str(hash(game\_hash))

            Saved\_Game.cookie\_key == str(cookie\_key)

        )\

        .delete()

    # create a new entry in the database and commit

    session.add(

        Saved\_Game(game, cookie\_key)

    )

    session.commit()

It contains a pair of function that are able to save a Game\_Website object to the database as binary and the query and deserialize it to restore the Game\_Website object.

I have used the pickle library to convert my python object to and from binary.

We can also see that I can execute SQL queries in a pythonic way thanks to the use of an ORM framework. This adds validation to prevent SQL injection attacks. This is because, whatever the malicious code stored in the cookie, it will not be executed an will be treated as a sting due to the type definition in the Saved\_Game class.

Schema Module:

The schema module is responsible for serialisation and serialisation processes. This is used to serialise python objects’ data into a form that can be stored in a database or sent to a client’s browser. The python objects can also be restored using the corresponding deserialization functions if needed.

The schema module’s \_\_init\_\_.py file exports for following functions:

from .cache\_item\_schema import minimax\_cache\_item\_schema

from .socket\_schemas import serialize\_legal\_moves, serialize\_pieces\_matrix, deserialize\_move, serialize\_piece, serialize\_move, deserialize\_pieces\_matrix

Here is cache\_item\_schema.py:

# import external and external modules

from marshmallow import Schema, fields, pre\_load, pre\_dump, post\_load, post\_dump

from chess\_functions import Vector

from database import Minimax\_Cache\_Item

# the vector schema describes how to serialize and deserialize a vector object

# it converts between Vector(i=a, j=b) and [a, b]

class Vector\_Schema(Schema):

    i = fields.Integer(required=True)

    j = fields.Integer(required=True)

    @pre\_load

    def pre\_load(self, vector, \*\*kwargs):

        data = {}

        data["i"] = vector.i

        data["j"] = vector.j

        return data

    @post\_dump

    def post\_dump(self, data, \*\*kwargs):

        return Vector(

            i=data["i"],

            j=data["j"]

        )

# this move schema is for database cache

# it converts between [Vector(a, b), Vector(c, d)] and [[a,b], [c,d]]

# is used the nested Vector Schema to do this

class Move\_Schema(Schema):

    from\_vector = fields.Nested(Vector\_Schema, required=True)

    movement\_vector = fields.Nested(Vector\_Schema, required=True)

    @pre\_load

    def pre\_load(self, vector\_tuple, \*\*kwargs):

        # print({"vector\_tuple": vector\_tuple})

        from\_vector, movement\_vector = vector\_tuple

        data = {}

        data["from\_vector"] = from\_vector

        data["movement\_vector"] = movement\_vector

        return data

    @post\_dump

    def post\_dump(self, data, \*\*kwargs):

        return (

            data["from\_vector"],

            data["movement\_vector"]

        )

# this method deserializes a move string into a pair of Vectors stored in a dictionary

def get\_deserialized\_move(move):

    # assert len(move) == 4

    # if move == "None":

    if move == None:

        return None

    move = move[1:-1]

    move = move.split(", ")

    from\_vector = Vector(

        i=move[0],

        j=move[1]

    )

    movement\_vector = Vector(

        i=move[2],

        j=move[3]

    )

    return dict(

        from\_vector=from\_vector,

        movement\_vector=movement\_vector

    )

# this method converts a pair of vectors stored in a dictionary to a string representation of the move

def get\_serialised\_move(move):

    if move is None:

        # return "None"

        return None

    return str((

        move["from\_vector"]["i"],

        move["from\_vector"]["j"],

        move["movement\_vector"]["i"],

        move["movement\_vector"]["j"],

    ))

# this class uses the move schema and the above methods to serialize and deserialize a move so that it can be stored in a database

class Minimax\_Cache\_Item\_Schema(Schema):

    board\_state\_hash = fields.String(required=True, load\_only=True)

    depth = fields.Integer(required=True)

    score = fields.Integer(required=True)

    move = fields.Nested(Move\_Schema, required=True, allow\_none=True)

    @post\_load

    def post\_load(self, data, \*\*kwargs):

        # assert isinstance(data["depth"], int)

        # assert isinstance(data["score"], int)

        # print(f"post\_load: serializing move form {data['move']} to {get\_serialised\_move(data['move'])}")

        return Minimax\_Cache\_Item(

            board\_state\_hash=data["board\_state\_hash"],

            depth=data["depth"],

            score=data["score"],

            move=get\_serialised\_move(

                data["move"]

            )

        )

    @pre\_dump

    def pre\_dump(self, minimax\_cache\_item: Minimax\_Cache\_Item, \*\*kwargs):

        # print(f"pre\_dump: deserializing move form {minimax\_cache\_item.move} to {get\_deserialized\_move(minimax\_cache\_item.move)}")

        return dict(

            depth=minimax\_cache\_item.depth,

            move=get\_deserialized\_move(

                minimax\_cache\_item.move

            ),

            score=minimax\_cache\_item.score

        )

# this object can be used to serialize and deserialize moves to be stored in the database

# it is the final product of this file and the object that will be exported

minimax\_cache\_item\_schema = Minimax\_Cache\_Item\_Schema()

This code will allow me to convert a dictionary description of all the properties of a minimax search to a Minimax\_Cache\_Item object that can be used as an entry in a database.

I also use schemas to serialise data for sending to the client:

Socket\_shemas.py:

from marshmallow import Schema, fields, pre\_load, pre\_dump, post\_load, post\_dump

from chess\_functions import Vector, Piece, PIECE\_TYPES

# different schema to serialize and deserialize a vector

class Vector\_Schema(Schema):

    i = fields.Integer(required=True)

    j = fields.Integer(required=True)

    @post\_load

    def get\_vector\_from\_internal\_data(self, internal\_data, \*\*kwargs):

        # print({"internal\_data": internal\_data})

        return Vector(\*\*internal\_data)

    @pre\_load

    def get\_internal\_data\_from\_list(self, list\_data, \*\*kwargs):

        return {"i": list\_data[0], "j": list\_data[1]}

    @post\_dump

    def make\_list\_from\_internal\_data(self, internal\_data, \*\*kwargs):

        return [internal\_data['i'], internal\_data['j']]

letter\_symbol\_pairs = (

    ("P", "♟︎"),

    ("K", "♚"),

    ("Q", "♛"),

    ("R", "♜"),

    ("B", "♝"),

    ("N", "♞"),

)

# these functions get the symbol or letter associated with each piece by searching the above tuple

def get\_symbol\_from\_letter(target\_letter):

    for letter, symbol in letter\_symbol\_pairs:

        if letter == target\_letter.strip().upper():

            return symbol

    raise ValueError(f"letter  '{letter}'  not found")

def get\_letter\_from\_symbol(target\_symbol):

    for letter, symbol in letter\_symbol\_pairs:

        if symbol == target\_symbol.strip():

            return letter

    raise ValueError(f"symbol  '{symbol}'  not found")

# this schema serialised and deserializes between the Pieces object and format use in JSON sent to the client

class Piece\_Schema(Schema):

    color = fields.String(required=True)

    letter = fields.String(required=True)

    @pre\_load

    def pre\_load(self, data, \*\*kwargs):

        color, symbol = data

        letter = get\_letter\_from\_symbol(symbol)

        return {

            "color": color,

            "letter": letter,

        }

    @post\_load

    def post\_load(self, data, \*\*kwargs):

        color, letter = data["color"], data["letter"]

        Piece\_Type = PIECE\_TYPES[letter]

        return Piece\_Type(color)

    @pre\_dump

    def pre\_dump(self, piece: Piece, \*\*kwargs):

        color, letter = piece.symbol()

        return {

            "color": color,

            "letter": letter,

        }

    @post\_dump

    def post\_dump(self, data, \*\*kwargs):

        color, letter = data["color"], data["letter"]

        symbol = get\_symbol\_from\_letter(letter)

        return [color, symbol]

# these function are utility function that can map a function across a 1d or 2d array, transforming all the elements

def list\_map(some\_function, array):

    return list(map(some\_function, array))

def two\_d\_list\_map(some\_function, two\_d\_array):

    return list\_map(

        lambda one\_d\_array: list\_map(

            some\_function,

            one\_d\_array

        ),

        two\_d\_array

    )

# create objects form the schemas with which to serialize and deserialize

vector\_schema = Vector\_Schema()

piece\_schema = Piece\_Schema()

# here are functions that use thee schemas to serialize and deserialize key data points to and from a dictionary (JSON like) format

def serialize\_legal\_moves(legal\_moves):

    return two\_d\_list\_map(vector\_schema.dump, legal\_moves)

def deserialize\_legal\_moves(legal\_moves):

    return two\_d\_list\_map(vector\_schema.load, legal\_moves)

def serialize\_move(move):

    return list\_map(vector\_schema.dump, move)

def deserialize\_move(move):

    return list\_map(vector\_schema.load, move)

def serialize\_piece(piece):

    if piece is None:

        return [None, None]

    else:

        return piece\_schema.dump(piece)

def deserialize\_piece(piece):

    # print(piece, end=";   ")

    if piece == [None, None]:

        return None

    else:

        return piece\_schema.load(piece)

def serialize\_pieces\_matrix(pieces\_matrix):

    return two\_d\_list\_map(serialize\_piece, pieces\_matrix)

def deserialize\_pieces\_matrix(pieces\_matrix):

    # print("deserialize\_pieces\_matrix   called")

    return two\_d\_list\_map(deserialize\_piece, pieces\_matrix)

Here is a jupyter notebook where I show what each of these schemas does:

Graphical user interface, text

Description automatically generated

Text

Description automatically generated with medium confidence

Graphical user interface, application

Description automatically generated

These functions will be used as shown to format data form python objects to data structures like dictionaries that can be easily converted to JSON.

Website Module:

This module is responsible for creating a webserver to how the webpage and for handling all the client page’s requests.

Here is the \_\_init\_\_.py file:

from .flask\_server import app

As we can see, the only component that is exported is called app. This is a flask object that has been given all the functionality we want in out webserver / socket server.

The website is run by a file in the root directory called app.py:

from website import app

from database import persistent\_DB\_engine, volatile\_RAM\_engine, end\_engines

# from geventwebsocket.server import WSGIServer

# import socketio

# import flask

host = '127.0.0.1'

port\_num = 5000

url = f"http://{host}:{port\_num}"

def run\_app():

    print(url)

    try:

        app.run(

            host=host,

            port=port\_num,

            debug=True

        )

    except KeyboardInterrupt:

        end\_engines([persistent\_DB\_engine, volatile\_RAM\_engine])

# def run\_app():

#     print(url)

#     try:

#         socketio.run(app)

#     except KeyboardInterrupt:

#         pass

# def run\_app():

#     http\_server = WSGIServer(('0.0.0.0', 5000), app)

#     http\_server.serve\_forever()

# def run\_app():

#     # socketio.run(app, host='0.0.0.0', port=5000, debug=True, use\_reloader=False)

#     # flask.run(app, host='0.0.0.0', port=5000, debug=True, use\_reloader=False)

#     app.run(host=host, port=port\_num, debug=True, use\_reloader=False)

# use python -m flask run to stop it raising warnings about how it should be deployed

if \_\_name\_\_ == "\_\_main\_\_":

    run\_app()

from website import app

from database import persistent\_DB\_engine, volatile\_RAM\_engine, end\_engines

# variables to decide the URL of the website

host = '127.0.0.1'

port\_num = 5000

url = f"http://{host}:{port\_num}"

# this function runs the flask server (hosting the website)

# to close the server, use ctrl + c to create a keyboard interrupt,

# this will safely destroy all engine connections to the database

def run\_app():

    print(url)

    try:

        app.run(

            host=host,

            port=port\_num,

            debug=True

        )

    except KeyboardInterrupt:

        end\_engines([persistent\_DB\_engine, volatile\_RAM\_engine])

# use python -m flask run to stop it raising warnings about how it should be deployed

if \_\_name\_\_ == "\_\_main\_\_":

    run\_app()

the website is run with the following command:

Text

Description automatically generated

The python file flask\_server.py contains all the server side logic of the webserver and socket server:

# import external modules

import flask

import os

from flask\_socketio import SocketIO

import datetime

import pickle

# import local modules

from assorted import safe\_hash

from database import save\_game, get\_saved\_game, persistent\_DB\_engine, create\_session, end\_session

from chess\_game import Game\_Website

from schemas import serialize\_legal\_moves, deserialize\_move, serialize\_pieces\_matrix, serialize\_piece

# fix tabes don't exist on reload bug: https://stackoverflow.com/questions/44531360/flask-blogging-error-table-doesnt-exist-tables-not-being-created

# establish various important directories in variables

basedir = os.getcwd()

static\_folder\_path = os.path.join(basedir, 'website', 'static')

template\_folder\_path = os.path.join(basedir, 'website', 'templates')

key\_file\_path = os.path.join(basedir, 'website', 'secret\_key.key')

# retrieve the secret key

with open(key\_file\_path, "r") as file:

    secret\_key = file.read()

# sanitize secret key to remove white space

secret\_key = secret\_key\

    .replace(" ", "")\

    .replace("\n", "")\

    .replace(chr(13), "")

# create an app object and configure it

app = flask.Flask(

    \_\_name\_\_,

    static\_folder=static\_folder\_path,

    template\_folder=template\_folder\_path,

)

# app.config['SEND\_FILE\_MAX\_AGE\_DEFAULT'] = 0

app.secret\_key = secret\_key

# create the socketio object (bound to the app object)

# socketio = SocketIO(app, async\_mode=None)

# socketio = SocketIO(app, async\_mode="gevent")

socketio = SocketIO(app, async\_mode=None, threaded=True, echo=False)

@app.route("/")

def index():

    # this is the only http endpoint,

    # it renders the appropriate html file

    response = flask.make\_response(

        flask.render\_template("chess\_game.html")

    )

    # ot then adds a cookie to the response that contains a cookie key corresponding to the game in the session

    expires = datetime.datetime.now() + datetime.timedelta(hours=48)

    cookie\_key = flask.session["cookie\_key"]

    cookie\_formatted = str(cookie\_key).encode("utf-8")

    response.set\_cookie(

        "chess\_game\_cookie\_key",

        cookie\_formatted,

        expires=expires

    )

    # print(f"index: cookie created: {cookie\_key}")

    # the response is returned, creating the cookie and providing the html file for the webpage

    return response

# the below functions both get and save a Game\_Website object to the flask session as a binary blob

# this prevent issues that can occur when python objects are saved directly in the flask session

def get\_game\_in\_session() -> Game\_Website:

    # print(repr(flask.session["game"]))

    # print(repr(pickle.loads(flask.session["game"])))

    return pickle.loads(

        flask.session["game"]

    )

def set\_game\_in\_session(game: Game\_Website):

    flask.session["game"] = pickle.dumps(

        game

    )

# def get\_game\_in\_session() -> Game\_Website:

#     return flask.session["game"]

# def set\_game\_in\_session(game: Game\_Website):

#     flask.session["game"] = game

# this handler function is responsible for initializing a flask session

def create\_flask\_session():

    # if flask.session.get("session\_setup\_complete"):

    #     return None

    # else:

    #     flask.session["session\_setup\_complete"] = True

    # it first checks the cookies

    # print("Creating session")

    # check cookies

    # game\_hash\_cookie = flask.request.cookies.get("chess\_game\_hash")

    game\_cookie = flask.request.cookies.get("chess\_game\_cookie\_key")

    # print(f"In creating session, cookie chess\_game\_cookie\_key fetched, it contains:   {game\_cookie}")

    # if the appropriate cookie doesn't exist, a random number is used as the cookie key,

    # and a fresh game is created

    if game\_cookie is None:

        print("Cookie was none / doesn't exist")

        game = Game\_Website()

        cookie\_key = safe\_hash(os.urandom(32))

    # if a cookie id was recovered

    else:

        # the id is converted into a python sting that contains hexadecimal characters

        hex\_string = game\_cookie.encode("utf-8").hex()

        cookie\_key = bytes.fromhex(hex\_string).decode('utf-8')

        print(f"flask cookie contained hash: {cookie\_key}")

        # a database session is created

        session\_DB = create\_session(persistent\_DB\_engine)

        # the game object is retrieved from the database

        # database\_lookup\_result: Game\_Website = get\_saved\_game(game\_hash=game\_hash, session=session)

        database\_lookup\_result: Game\_Website = get\_saved\_game(cookie\_key=cookie\_key, session=session\_DB)

        # print("database\_lookup\_result")

        # print(database\_lookup\_result)

        # the session is disposed of

        end\_session(session\_DB)

        session\_DB = None

        # if the game was found in the database, use it, else create a new game

        if database\_lookup\_result is None:

            print(f"game: NOT successfully read from database (returned None)")

            game = Game\_Website()

        else:

            print(f"game successfully read from database")

            game = database\_lookup\_result

            print(f"game with hash {hash(game)} successfully read from database")

    # now that a game object and a cookie\_key have been determined, store these in the session object

    # the index http route will create the appropriate cookie using this id

    # print(repr(game))

    set\_game\_in\_session(game)

    # print(f"At the end of create\_flask\_session, setting cookie\_key to :  {cookie\_key}")

    flask.session["cookie\_key"] = cookie\_key

    # print(f"At the end of create\_flask\_session, game used has ID:  {hash(game)}")

# this handler function is responsible for clearing up a flask session

# this involves saving the game to the database

def close\_flask\_session():

    # if flask.session.get("session\_setup\_complete"):

    #     flask.session["session\_setup\_complete"] = False

    # else:

    #     return None

    # print("Closing session")

    # game: Game\_Website = get\_game\_in\_session()

    # get tha game from the database

    game: Game\_Website = pickle.loads(

        flask.session["game"]

    )

    # if the window was closed part way though the computer move

    # quickly decide the computers move (time=0) so that game can be resorted on the user's turn

    if game.board\_state.next\_to\_go == "B":

        game.implement\_computer\_move\_and\_report(0)

    # print("close\_flask\_session, game in session is: ")

    # game.board\_state.print\_board()

    # game = session["game"]

    # get the cookie id,

    cookie\_key = flask.session["cookie\_key"]

    # make a database session and save the game to the database using the cookie id

    session\_DB = create\_session(persistent\_DB\_engine)

    save\_game(game=game, cookie\_key=cookie\_key, session=session\_DB)

    end\_session(session\_DB)

    session\_DB = None

@app.before\_request

def handle\_create\_session(\*args, \*\*kwargs):

    # this runs the handler function to create a session before the first request (using the session created flag)

    if not flask.session.get("session\_created"):

        flask.session["session\_created"] = True

        # print("before\_request, session:")

        # print(flask.session)

        return create\_flask\_session()

# unfortunately I needed to use 2 separate events to fully close the flask session

# the on socket disconnect event allows me to access the session data but not change it

# the on http request stop\_and\_save\_game event runs after the session data is deleted,

# it allow me to change the session data and set the session created flag to false

@socketio.on("disconnect")

def handle\_close\_session(\*args, \*\*kwargs):

    # this runs the handler to dispose of the session when the socket connection breaks

    close\_flask\_session()

@app.route('/stop\_and\_save\_game')

def handle\_close\_session(\*args, \*\*kwargs):

    # this function sets the flag to indicate that the session needs to be reinitialized

    flask.session["session\_created"] = False

    return "session closed"

# here I have created a socket handler function

# it is a decorator as it takes a function as a parameter and returns a modified function with additional functionality

def bind\_socket\_handler(event\_name, respond=True):

    def decorator(function):

        request\_event = f"{event\_name}\_request"

        response\_event = f"{event\_name}\_response"

        # new function to be returned takes incoming payload as an argument

        def wrapper(incoming\_payload: dict | None = None):

            # print(f"Handling event: {request\_event}")

            # if there was an incoming payload, this is given to the original function as an argument after being deserialized

            if incoming\_payload is None:

                result = function()

            else:

                # print({"incoming\_payload": incoming\_payload})

                if not isinstance(incoming\_payload, dict):

                    raise TypeError(f"incoming payload of unexpected type:   {type(incoming\_payload)}")

                result = function(incoming\_payload)

            # if the respond flag is true, the return value of the function is send back to the client

            if respond:

                outgoing\_payload: dict = result

                # print(f"Sending response payload by event {response\_event}")

                # print({"outgoing\_payload": outgoing\_payload})

                if not isinstance(outgoing\_payload, dict):

                    raise TypeError(f"outgoing payload of unexpected type:   {type(outgoing\_payload)}")

                socketio.emit(response\_event, outgoing\_payload)

        # the wrapper function name is set to the name of the original function

        wrapper.\_\_name\_\_= function.\_\_name\_\_

        # bind wrapper function to happen when request corresponding to the event is received

        # use decorator in manual way to not overwrite wrapper

        socketio.on(request\_event)(wrapper)

        # return wrapper for any further use

        return wrapper

    return decorator

# the below function accesses the game object in the session

# it returns a dictionary of serialized data about this game object

def generate\_game\_update\_data():

    game: Game\_Website = get\_game\_in\_session()

    # collect various data point from the game object and serialize as necessary

    next\_to\_go = game.board\_state.next\_to\_go

    difficulty = game.difficulty

    legal\_moves = list(game.board\_state.generate\_legal\_moves())

    legal\_moves\_serialised = serialize\_legal\_moves(legal\_moves)

    pieces\_matrix = game.board\_state.pieces\_matrix

    pieces\_matrix\_serialised = serialize\_pieces\_matrix(pieces\_matrix)

    next\_to\_go\_in\_check = game.board\_state.color\_in\_check()

    over, winning\_player, victory\_classification = game.check\_game\_over()

    game\_over\_data =  {

        "over": over,

        "winning\_player": winning\_player,

        "victory\_classification": victory\_classification,

    }

    pieces\_missing = {}

    for color in ("B", "W"):

        pieces\_missing[color] = list(map(

            serialize\_piece,

            game.board\_state.generate\_pieces\_taken\_by\_color(color)

        ))

    move\_history = game.move\_history\_output

    # this is the format of the outgoing payload,

    # flask will convert it to JSON automatically

    # this payload contains all the data that the client needs to be updated about the game state

    payload = {

        "difficulty": difficulty,

        "next\_to\_go":  next\_to\_go,

        "game\_over\_data":  game\_over\_data,

        "legal\_moves":  legal\_moves\_serialised,

        "pieces\_matrix": pieces\_matrix\_serialised,

        "check": next\_to\_go\_in\_check,

        "pieces\_taken": pieces\_missing,

        "move\_history": move\_history,

    }

    return payload

# using my custom socket decorator greatly simplifies the process of using sockets

# on request get update from the client, return the generic payload of game data

@bind\_socket\_handler("get\_update")

def get\_update():

    """Sends all data from game object to client to update chess game

    This data is also updated after the user move and after the computer move"""

    result = generate\_game\_update\_data()

    # print(f"Update payload for game with hash  {hash(get\_game\_in\_session())}")

    # print(result)

    return result

# on request to implement the computer move

@bind\_socket\_handler("implement\_computer\_move")

def implement\_computer\_move():

    # get the game and validate that the computer can go

    game = get\_game\_in\_session()

    assert game.board\_state.next\_to\_go == "B"

    assert not game.board\_state.is\_game\_over\_for\_next\_to\_go()[0]

    # generate the move and a dictionary of data about it for the client

    # print("generating move")

    move\_description = game.implement\_computer\_move\_and\_report()

    # update the session with the new mutated game object

    set\_game\_in\_session(game)

    # print("after computer move: game in session is:")

    # get\_game\_in\_session().board\_state.print\_board()

    # generate a generic update payload and add teh move description before returning

    outgoing\_payload = generate\_game\_update\_data()

    outgoing\_payload["computer\_move\_description"] = move\_description

    # print(outgoing\_payload)

    return outgoing\_payload

# when a request comes to reset the game

# (no response needed)

@bind\_socket\_handler("reset\_game", respond=False)

def reset\_game():

    # preserve the difficulty setting

    old\_game = get\_game\_in\_session()

    # difficulty preserved

    difficulty = old\_game.difficulty

    # create a new game object with the old difficulty and update the session

    new\_game = Game\_Website(difficulty=difficulty)

    set\_game\_in\_session(new\_game)

    # return generate\_game\_update\_data()

# takes input from client about the user's move

@bind\_socket\_handler("implement\_user\_move")

def implement\_user\_move(incoming\_payload):

    game = get\_game\_in\_session()

    # validate that the user can move

    assert game.board\_state.next\_to\_go == "W"

    assert not game.board\_state.is\_game\_over\_for\_next\_to\_go()[0]

    # print("python function called: implement\_user\_move")

    # deserialize the user's move into vectors

    user\_move = tuple(

        deserialize\_move(

            incoming\_payload["user\_move"]

        )

    )

    # implement the move on the game object

    game.implement\_user\_move(user\_move)

    # print("python function finished: implement\_user\_move")

    # update the session with the new mutated game object

    set\_game\_in\_session(game)

    # print("after user move: game in session is:")

    # get\_game\_in\_session().board\_state.print\_board()

    # generate and return an update payload

    return generate\_game\_update\_data()

# on request to change difficulty

@bind\_socket\_handler("change\_difficulty", respond=False)

def change\_difficulty(incoming\_payload):

    game = get\_game\_in\_session()

    # get the new difficulty form the payload

    new\_difficulty = incoming\_payload["new\_difficulty"]

    # print(f"Changing difficulty to {new\_difficulty}")

    # mutate the game object and save it to the session

    game.difficulty = new\_difficulty

    set\_game\_in\_session(game)

    # doesn't return an update

    # return generate\_game\_update\_data()

# this code below was used to generate the initial game data json file that I no longer use

# before the reload game feature, this prevented the client needing to request the initial game data immediately on loading

# app = None

# if \_\_name\_\_ == "\_\_main\_\_":

#     import json

#     game: Game\_Website = Game\_Website()

#     next\_to\_go = game.board\_state.next\_to\_go

#     difficulty = game.difficulty

#     legal\_moves = list(game.board\_state.generate\_legal\_moves())

#     legal\_moves\_serialised = serialize\_legal\_moves(legal\_moves)

#     pieces\_matrix = game.board\_state.pieces\_matrix

#     pieces\_matrix\_serialised = serialize\_pieces\_matrix(pieces\_matrix)

#     next\_to\_go\_in\_check = game.board\_state.color\_in\_check()

#     over, winning\_player, victory\_classification = game.check\_game\_over()

#     game\_over\_data =  {

#         "over": over,

#         "winning\_player": winning\_player,

#         "victory\_classification": victory\_classification,

#     }

#     pieces\_missing = {}

#     for color in ("B", "W"):

#         pieces\_missing[color] = list(map(

#             serialize\_piece,

#             game.board\_state.generate\_pieces\_taken\_by\_color(color)

#         ))

#     move\_history = game.move\_history\_output

#     payload = {

#         "difficulty": difficulty,

#         "next\_to\_go":  next\_to\_go,

#         "game\_over\_data":  game\_over\_data,

#         "legal\_moves":  legal\_moves\_serialised,

#         "pieces\_matrix": pieces\_matrix\_serialised,

#         "check": next\_to\_go\_in\_check,

#         "pieces\_taken": pieces\_missing,

#         "move\_history": move\_history,

#     }

#     with open("./website/static/initial\_game\_data.json", "w") as file:

#         file.write(

#             json.dumps(payload)

#         )

As explained in the comments, the above code contains:

* A http route to set a cookie and give the webpage html
* Many socket event handlers to allow the client and the server to communicate
* Functions for saving the game in the session to the database and reloading it into the session from the id in the client’s cookie.

Due to the fact that I couldn’t set cookies to the client browser in any method other that the index http handler (only http request direct from browser) I needed to decide the cookie key that I would use before the final game was saved. This means that I had to use a large random number and couldn’t use the hash of the game state. This is because the handle session close method couldn’t set cookies.

I then used client side HTML, CSS and JavaScript for the front end component of the website:

Chess\_game.html:

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta http-equiv="X-UA-Compatible" content="IE=edge">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Chess Game</title>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.5.1/socket.io.js"

        integrity="sha512-9mpsATI0KClwt+xVZfbcf2lJ8IFBAwsubJ6mI3rtULwyM3fBmQFzj0It4tGqxLOGQwGfJdk/G+fANnxfq9/cew=="

        crossorigin="anonymous" referrerpolicy="no-referrer"></script>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/crypto-js/4.1.1/crypto-js.min.js"

        integrity="sha512-E8QSvWZ0eCLGk4km3hxSsNmGWbLtSCSUcewDQPQWZF6pEU8GlT8a5fF32wOl1i8ftdMhssTrF/OhyGWwonTcXA=="

        crossorigin="anonymous" referrerpolicy="no-referrer"></script>

    <!-- <script type="application/json" src="{{  url\_for('static', filename='initial\_game\_data.json')  }}"></script> -->

    <script type="text/javascript" src="{{  url\_for('static', filename='initial\_game\_data.js')  }}"></script>

    <script type="text/javascript" src="{{  url\_for('static', filename='vector.js')  }}"></script>

    <script type="text/javascript" src="{{  url\_for('static', filename='chess\_class.js')  }}"></script>

    <link rel="stylesheet" href="{{  url\_for('static', filename='style.css')  }}">

    <link rel="shortcut icon" href="{{  url\_for('static', filename='favicon.ico')  }}" type="image/x-icon">

</head>

<body>

<div class="centered">

        <h1 id="top\_title"></h1>

        <!-- <span class="spacer" style="height: 1vmin;"></span> -->

        <span class="spacer"></span>

        <div id="board"></div>

        <!-- <span class="spacer" style="height: 1vmin;"></span> -->

        <span class="spacer"></span>

        <div class="option\_button\_container">

            <!-- <button class="option\_button" id="concede\_button">Concede Game</button> -->

            <button class="option\_button" id="concede\_button">Concede</button>

            <!-- <button class="option\_button" id="restart\_button">Restart Game</button> -->

            <button class="option\_button" id="restart\_button">Restart</button>

        </div>

        <!-- <div class="spacer"></div> -->

        <span class="spacer"></span>

        <span class="spacer"></span>

        <div class="difficulty\_form">

            <h1>Change Difficulty: (AI thinking time)</h1>

            <!-- <p>Change Difficulty:</p> -->

            <form id="difficulty\_form">

                <label>

                    <input type="radio" name="difficulty" value="really\_easy" onclick="handle\_radio\_button\_click(this)">

                    Trivial (1 sec)

                </label>

                <label>

                    <input type="radio" name="difficulty" value="easy" onclick="handle\_radio\_button\_click(this)">

                    Easy (2 sec)

                </label>

                <label>

                    <input type="radio" name="difficulty" value="medium" onclick="handle\_radio\_button\_click(this)">

                    Medium (5 sec)

                </label>

                <br>

                <label>

                    <input type="radio" name="difficulty" value="hard" onclick="handle\_radio\_button\_click(this)">

                    Hard (10 sec)

                </label>

                <label>

                    <input type="radio" name="difficulty" value="really\_hard" onclick="handle\_radio\_button\_click(this)">

                    Challenge (15 sec)

                </label>

                <label>

                    <input type="radio" name="difficulty" value="extreme" onclick="handle\_radio\_button\_click(this)">

                    Extreme (30 sec)

                </label>

                <br>

                <label>

                    <input type="radio" name="difficulty" value="legendary" onclick="handle\_radio\_button\_click(this)">

                    Legendary (60 sec)

                </label>

            </form>

        </div>

        <!-- <div class="spacer"></div> -->

        <span class="spacer"></span>

        <span class="spacer"></span>

        <div class="pieces\_taken\_table">

            <h1>Pieces Taken:</h1>

            <table>

                <tr>

                    <td>White Pieces Taken</td>

                    <td>Black Pieces Taken</td>

                </tr>

                <tr>

                    <th id="num\_pieces\_taken\_white"></th>

                    <th id="num\_pieces\_taken\_black"></th>

                </tr>

                <tr>

                    <th id="which\_pieces\_taken\_white">-</th>

                    <th id="which\_pieces\_taken\_black">-</th>

                </tr>

            </table>

        </div>

        <!-- <div class="spacer"></div> -->

        <span class="spacer"></span>

        <span class="spacer"></span>

        <div class="previous\_moves\_table">

            <h1>Moves History:</h1>

            <table id="pieces\_taken\_table\_table\_tag">

                <tr>

                    <td>

                        White previous moves:

                    </td>

                    <td>

                        Black previous moves:

                    </td>

                </tr>

                <!-- <tr>

                    <td>

                        white move

                    </td>

                    <td>

                        black move

                    </td>

                </tr> -->

            </table>

        </div>

        <span class="spacer"></span>

        <span class="spacer"></span>

</div>

<script type="text/javascript" src="{{  url\_for('static', filename='main.js')  }}"></script>

</body>

</html>

The above html contains some bare bones tags with ID’s that allow them to be accessed by JS DOM (document object model) manipulator methods.

The chess board is drawn by JavaScript.

src="{{ url\_for('static', filename='main.js') }}"

The strange source and href attributes of script and style tags that link to external files will be overwritten with the dynamic URL of the file by the flak server when the HTML is rendered.

Here is main.js:

// these constants define the colors of text, squares and text shadow throughout the program

const white\_sq\_bg\_color = '#f5e6bf';

// const white\_sq\_bg\_color = '#d9cba7';

const black\_sq\_bg\_color = '#66443a';

const white\_piece\_color = '#ffffff';

const black\_piece\_color = '#000000';

// const white\_shadow = '-1px -1px 0 #000, 1px -1px 0 #000, -1px 1px 0 #000, 1px 1px 0 #000'

const black\_shadow = '-1px -1px 0 #fff, 1px -1px 0 #fff, -1px 1px 0 #fff, 1px 1px 0 #fff'

const white\_shadow = '-2px -2px 0 #000, 2px -2px 0 #000, -2px 2px 0 #000, 2px 2px 0 #000'

// const black\_shadow = '-2px -2px 0 #fff, 2px -2px 0 #fff, -2px 2px 0 #fff, 2px 2px 0 #fff'

// this function created a series of div tags that represent the individual squared in the chess board

// it decides their color, position vector and hence their ID and click event functions

function create\_board\_widget() {

    let board = document.getElementById("board");

    for (let row = 0; row <= 7; row++) {

        for (let col = 0; col <= 7; col++) {

            let i = col;

            let j = 7-row;

            // if row and column are both odd or both even then white, so add together and check if even

            let sum\_is\_even = ((row + col) % 2 == 0);

            // console.log(`square (${i},${j}), sum is even -->  ${sum\_is\_even}`)

            let square\_bg\_color = sum\_is\_even ? white\_sq\_bg\_color : black\_sq\_bg\_color;

            let square = document.createElement("div");

            // square.textContent = `(${i},${j})`;

            square.id = `square\_${i}${j}`;

            square.classList.add("square");

            square.style.backgroundColor = square\_bg\_color;

            square.addEventListener("click", function(){handle\_square\_click(i, j)})

            board.append(square)

        }

    }

}

// this function returns the html element of a square given a position vector

function get\_square\_at\_vector(v) {

    let [i, j] = v;

    let id = `square\_${i}${j}`;

    let square = document.getElementById(id);

    return square;

}

// this function uses the pieces matrix from the board object to populate the board and its squares with pieces

// it must decide their color and text shadow as well

function add\_pieces(board) {

    let pieces\_matrix = board.pieces\_matrix

    // iterate through rows and columns

    for (let row = 0; row <= 7; row++) {

        for (let col = 0; col <= 7; col++) {

            // get the vector and decide the color

            let [i, j] = [col, 7-row]

            let [color\_char, symbol] = pieces\_matrix[row][col];

            let color = (color\_char == "W") ? white\_piece\_color : black\_piece\_color;

            // console.log(`get\_square\_at\_vector([i, j])   --->   get\_square\_at\_vector(${[i, j]})   --->   ${get\_square\_at\_vector([i, j])}`)

            square = get\_square\_at\_vector([i, j]);

            // square.innerText = `(${i}, ${j})`;

            // square.style.fontSize = "20px"

            square.innerText = symbol;

            square.style.color = color;

            square.style.textShadow = (color\_char == "W") ? white\_shadow : black\_shadow;

            square.style.fontSize = "9vmin"

        }

    }

}

// this function uses the highlighted squares method of the board object to add highlighting to the chess board

// this highlights clicked pieces red and squares they can move to green

function add\_highlighting(board) {

    let new\_highlighting = board.get\_highlighted\_squares()

    // console.log(`add\_highlighting function called with ${new\_highlighting}`)

    // iterate through the position vectors and colors form the get\_highlighted\_squares method

    for (let i in new\_highlighting) {

        let [vector, color] = new\_highlighting[i];

        // console.log(`Adding ${color} to square at vector ${vector}`);

        square = get\_square\_at\_vector(vector);

        // console.log({ vector })

        // console.log({ square })

        // if (square.innerText == null) {

        // if a square should be highlighted but is empty: add a center dot

        if (square.innerText.trim() == "") {

            square.innerText = "·";

            // square.style.fontSize = "20vmin";

            square.style.fontSize = "10vmin";

        }

        square.style.color = color;

        // square.style.backgroundColor = color;

    }

}

// this function was not needed in the end

// if wipes the board of all pieces and highlighting

function clear\_board() {

    for (let i = 0; i <= 7; i++) {

        for (let j = 0; j <= 7; j++) {

            square = get\_square\_at\_vector([i, j]);

            // console.log(`Square at vector (${[i, j]}) is ${square}`);

            square.textContent = "";

            // square.innerText = "";

            square.style.color = "black";

            square.style.textShadow = "";

            square.style.fontSize = "9vmin"

        }

    }

}

// this function updates the main title widget with a new title based on the board object

function update\_main\_title(board, manual\_new\_title=null) {

    if (manual\_new\_title !== null) {

        document.getElementById("top\_title").innerText = manual\_new\_title

        return null;

    }

    let title\_msg

    if (board.just\_conceded) {

        // console.log({board})

        // console.log(JSON.stringify(board))

        // console.log(board.just\_conceded)

        title\_msg = "Game Over: You Conceded"

    }

    else {

        title\_msg = `${(board.next\_to\_go === "W") ? "Your Go:" : "Computer's Go... (please wait)"}${(board.check) ? "   (CHECK)" : ""}`

    }

    document.getElementById("top\_title").innerText = title\_msg

}

// this function adds the pieces taken data (number and specific pieces) to the table

function update\_pieces\_taken(board) {

    let num\_B\_taken = board.pieces\_taken.B.length

    let num\_W\_taken = board.pieces\_taken.W.length

    let B\_pieces\_conjugation = (num\_B\_taken == 1) ? "Piece": "Pieces"

    let W\_pieces\_conjugation = (num\_W\_taken == 1) ? "Piece": "Pieces"

    document.getElementById("num\_pieces\_taken\_black").innerText = (num\_B\_taken > 0) ? `${num\_B\_taken} ${B\_pieces\_conjugation} Taken`: "No Pieces Taken";

    document.getElementById("num\_pieces\_taken\_white").innerText = (num\_W\_taken > 0) ? `${num\_W\_taken} ${W\_pieces\_conjugation} Taken` : "No Pieces Taken";

    // document.getElementById("which\_pieces\_taken\_black").innerText = (board.pieces\_taken.B.length > 0) ? `Pieces Lost: ${board.pieces\_taken.B.map(x => x[1]).join(", ")}`: "-";

    document.getElementById("which\_pieces\_taken\_black").innerText = (board.pieces\_taken.B.length > 0) ? `${board.pieces\_taken.B.map(x => x[1]).join(", ")}`: "-";

    // document.getElementById("which\_pieces\_taken\_white").innerText = (board.pieces\_taken.W.length > 0) ? `Pieces Lost: ${board.pieces\_taken.W.map(x => x[1]).join(", ")}`: "-";

    document.getElementById("which\_pieces\_taken\_white").innerText = (board.pieces\_taken.W.length > 0) ? `${board.pieces\_taken.W.map(x => x[1]).join(", ")}`: "-";

}

// this function auto selects one of the difficulty radio buttons

function set\_selected\_difficulty(board) {

    let difficulty = board.difficulty

    // console.log({difficulty})

    document.querySelector(`input[value=${difficulty}]`).checked = true;

    // document.querySelector(`input[value="${difficulty}"]`).checked = true;

    // document.querySelector(`input[value=${difficulty}]`).checked = true;

}

// this function adds moves to the move history table

function set\_widget\_move\_history(board) {

    let moves = board.move\_history.length;

    let rows

    let move\_history\_output = Array.from(board.move\_history)

    // depending on the number of moves made in the game, extra cells may need to be added to make the table look right

    if (moves == 0) {

        rows = 1;

        move\_history\_output.push("-")

        move\_history\_output.push("-")

    }

    else if (moves % 2 == 0) {

        rows = moves / 2

    }

    else {

        rows = (moves + 1) / 2

        move\_history\_output.push("-")

    }

    // console.log({half\_moves\_rounded\_up});

    // have a row for each number in half\_moves\_rounded\_up

    // delete old elements from the table

    let old\_rows = Array.from(document.querySelectorAll('.temporary\_previous\_moves\_table\_item'));

    if (old\_rows.length > 0) {

        for (let i in old\_rows) {

            old\_rows[i].removeChild(old\_rows[i].firstChild);

            old\_rows[i].removeChild(old\_rows[i].firstChild);

            old\_rows[i].parentNode.removeChild(old\_rows[i]);

        }

    }

    // iterate through the rows and add new TD tags to add the new row to the table

    table = document.getElementById("pieces\_taken\_table\_table\_tag")

    // console.log({table})

    let white\_move

    let black\_move

    let new\_row

    let cells

    for (let r=0; r<rows; r++) {

        white\_move = move\_history\_output[2\*r];

        black\_move = move\_history\_output[1+ 2\*r];

        new\_row = document.createElement("tr")

        cells = [

            document.createElement("td"),

            document.createElement("td")

        ]

        cells[0].innerText = white\_move;

        cells[1].innerText = black\_move;

        new\_row.classList.add("temporary\_previous\_moves\_table\_item")

        cells[0].classList.add("small\_text")

        cells[1].classList.add("small\_text")

        new\_row.appendChild(cells[0]);

        new\_row.appendChild(cells[1]);

        table.appendChild(new\_row)

    }

}

// gets the hash of a given piece of data

// https://stackoverflow.com/questions/54701686/matching-cryptojs-sha256-with-hashlib-sha256-for-a-json

function get\_hash\_of\_data(data, stringify = true) {

    let data\_string

    if (stringify) {

        data\_string = JSON.stringify(data)

    }

    else {

        data\_string = data

    }

    return CryptoJS.SHA256(data\_string).toString(CryptoJS.enc.Base64);

}

// holds a table that keep track of the hashes of various items of data

// includes data item name as a key and current hash + function to find hash as values

previous\_data\_hashes = {

    pieces\_taken: [null, (board) => get\_hash\_of\_data(board.pieces\_taken)],

    move\_history: [null, (board) => get\_hash\_of\_data(board.move\_history)],

    piece\_layout: [null, (board) => get\_hash\_of\_data([board.pieces\_matrix, board.possible\_to\_vectors, board.selected\_from\_vector])],

    highlighting: [null, (board) => get\_hash\_of\_data([board.possible\_to\_vectors, board.selected\_from\_vector])],

    difficulty: [null, (board) => get\_hash\_of\_data(board.difficulty, stringify=false)]

};

// this function only runs the DOM update method given if the data to display changed

function update\_as\_necessary(board, update\_function, hashes\_table\_key) {

    // create a new hash

    // only if it is different from the old hash, then update the board and the hash in the data previous\_data\_hashes object

    // console.log(previous\_data\_hashes[hashes\_table\_key])

    let [old\_hash, compute\_hash] = previous\_data\_hashes[hashes\_table\_key];

    let new\_hash = compute\_hash(board);

    if (new\_hash !== old\_hash) {

        // console.log(`hashes are different, updating ${hashes\_table\_key}`)

        previous\_data\_hashes[hashes\_table\_key][0] = new\_hash

        update\_function(board)

    }

    // else {

    //     console.log(`hashes the same, NOT updating ${hashes\_table\_key}`)

    // }

}

// update all dom widgets as necessary, always update title

function update\_board\_widget(board) {

    // clear\_board();

    // console.log("board.pieces\_matrix")

    // console.log(board.pieces\_matrix)

    // add\_pieces(board);

    // add\_highlighting(board);

    // update\_main\_title(board);

    // set\_selected\_difficulty(board);

    // update\_pieces\_taken(board);

    // set\_widget\_move\_history(board);

    update\_main\_title(board);

    update\_as\_necessary(board, add\_pieces, "piece\_layout")

    update\_as\_necessary(board, add\_highlighting, "highlighting")

    update\_as\_necessary(board, set\_selected\_difficulty, "difficulty")

    update\_as\_necessary(board, update\_pieces\_taken, "pieces\_taken")

    update\_as\_necessary(board, set\_widget\_move\_history, "move\_history")

}

// calls teh appropriate square click function on the board and then updates the DOM

function handle\_square\_click(i, j) {

    // board is a global variable

    // console.log(`Square at (${i}, ${j}) has been clicked`);

    board.handle\_square\_click([i, j]);

    // console.log(board.pieces\_matrix);

    update\_board\_widget(board);

    // alert("about to clear board");

    // clear\_board();

}

// whole file loads after html

// create chess board html elemtent

create\_board\_widget();

// create board class

let board = new Chess\_Board(

    update\_board\_widget,

    update\_main\_title

);

// bind buttons to methods of the board class

document.getElementById("restart\_button").addEventListener("click", function () {

    board.reset\_game()

})

document.getElementById("concede\_button").addEventListener("click", function () {

    // update\_main\_title(null, "You Conceded The Game")

    board.concede\_game()

})

// bind a change to the radio buttons to a handler method in the board class

function handle\_radio\_button\_click(radio\_button) {

    let new\_difficulty = radio\_button.value;

    if (new\_difficulty !== board.difficulty) {

        // console.log(`Setting new difficulty to ${new\_difficulty}`)

        board.change\_difficulty(new\_difficulty);

    }

}

// before letting the window close, send the server a warning so it can save the game

// also produces a pop up box to prevent accidental closing of the game

window.onbeforeunload = () => fetch('/stop\_and\_save\_game');

The main function of this file is to:

* Define methods that can affect the DOM to display outputs (changes in the Chess\_Board object)
* Create the chess board widget.
* Create an instance of the Chess\_Board class that is held in global variable called board.
* Tie together button click events to the board objects handlers and also provide the board with functions needed to update the html widgets.

The other main JavaScript file is the chess\_class.js file:

// utility functions for use in the program

function arrays\_are\_equal(a1, a2) {

    return JSON.stringify(a1) == JSON.stringify(a2)

}

function two\_d\_array\_contains\_sub\_array(two\_d\_array, sub\_array) {

    for (let i in two\_d\_array) {

        if (arrays\_are\_equal(two\_d\_array[i], sub\_array)) {

            return true;

        }

    }

    return false;

}

function assert(condition, message) {

    if (!condition) {

        message = (message === null? "Assertion error": message);

        throw new Error(message);

    }

}

// socket object to manage socket connection

let socket = io();

// this is a factory function that takes an event and returns an asynchronous function

// the function will send a request to the server and then await and return the server's resonance

function async\_function\_factory\_send\_and\_receive(event\_name) {

    // returns an async function

    async function external\_get\_response(outgoing\_payload = null) {

        // this promise resolves then a response from the server is received, the response is given

        const server\_response\_promise = new Promise(function (resolve) {

            socket.on(`${event\_name}\_response`, function (data) {

                resolve(data);

            })

        });

        // the request is sent to the server, along with an outgoing payload if appropriate

        if (outgoing\_payload === null) {

            socket.emit(`${event\_name}\_request`);

        }

        else {

            socket.emit(

                `${event\_name}\_request`,

                outgoing\_payload

            );

        }

        // the server's response is then awaited

        let response\_payload = await server\_response\_promise

        // console.log({ response\_payload })

        // if the response contains any actual data then it is returned

        var is\_null = (response\_payload === null)

        var is\_empty\_sting = (response\_payload === "")

        var is\_empty\_dict = (Object.keys(response\_payload).length === 0)

        if (is\_null || is\_empty\_sting || is\_empty\_dict) {

            return null;

        }

        else {

            return response\_payload

        }

    // the async function with this behaviour is returned to be reused

    }

    return external\_get\_response

}

// define async functions to make requests to server

// simpler functions that don't need to send and data to the server

external\_get\_update = async\_function\_factory\_send\_and\_receive("get\_update");

external\_implement\_computer\_move\_and\_update = async\_function\_factory\_send\_and\_receive("implement\_computer\_move")

external\_reset\_game = async\_function\_factory\_send\_and\_receive("reset\_game")

// more complex functions that must crate an outgoing payload and send it to the server, then return the response

async function external\_implement\_user\_move\_and\_update(from\_vector, to\_vector) {

    movement\_vector = subtract\_vectors(to\_vector, from\_vector);

    outgoing\_payload = {

        "user\_move": [from\_vector, movement\_vector],

    };

    data\_exchange\_handler\_function = async\_function\_factory\_send\_and\_receive("implement\_user\_move")

    server\_data = await data\_exchange\_handler\_function(outgoing\_payload);

    return server\_data;

}

async function external\_change\_difficulty(new\_difficulty) {

    // console.log(`sending server new difficulty of ${new\_difficulty}`)

    outgoing\_payload = { "new\_difficulty": new\_difficulty }

    external\_reset\_game = async\_function\_factory\_send\_and\_receive("reset\_game")

    data\_exchange\_handler\_function = async\_function\_factory\_send\_and\_receive("change\_difficulty")

    await data\_exchange\_handler\_function(outgoing\_payload);

}

// this class contains all the data and behaviour of the client side chess game

class Chess\_Board {

    // before reload feature, constructor just used initial game data

    // constructor(update\_board\_widget\_function) {

    //     this.update\_board\_widget = function () {

    //         update\_board\_widget\_function(this)

    //     }

    //     this.update\_from\_server\_data(INITIAL\_GAME\_DATA)

    //     this.just\_reset = true

    //     this.just\_conceded = false

    // };

    // new constructor sets dom manipulation as properties

    // then makes a request to the server to determine it properties

    constructor(update\_board\_widget\_function, update\_main\_title\_widget) {

        this.update\_board\_widget = function () {

            update\_board\_widget\_function(this)

        }

        this.update\_main\_title\_widget = function (msg) {

            update\_main\_title\_widget(this, msg)

        }

        // this.update\_from\_server\_data(INITIAL\_GAME\_DATA)

        this.direct\_server\_grand\_update()

    };

    // this updates the board with the server's data but also sets reset flags to false

    async direct\_server\_grand\_update() {

        let server\_data = await external\_get\_update();

        this.update\_from\_server\_data(server\_data);

        this.just\_reset = false

        this.just\_conceded = false

        assert(this.next\_to\_go !== "B")

    }

    // this function takes server data and unpacks it,

    // the properties of the board object are then updated with this new server data

    // the board widget is then redrawn

    update\_from\_server\_data(server\_data, user\_input\_disabled = false) {

        this.possible\_to\_vectors = [];

        this.selected\_from\_vector = null;

        this.user\_input\_disabled = user\_input\_disabled

        this.pieces\_matrix = server\_data.pieces\_matrix;

        this.legal\_moves = server\_data.legal\_moves;

        this.next\_to\_go = server\_data.next\_to\_go;

        this.check = server\_data.check;

        this.just\_reset = false

        // console.log("server\_data")

        // console.log(server\_data)

        // console.log("server\_data.game\_over\_data")

        // console.log(server\_data.game\_over\_data)

        this.game\_over = server\_data.game\_over\_data.over;

        delete server\_data.game\_over\_data.over;

        this.over\_data = server\_data.game\_over\_data;

        this.pieces\_taken = server\_data.pieces\_taken;

        this.move\_history = server\_data.move\_history;

        this.difficulty = server\_data.difficulty;

        this.update\_board\_widget();

    }

    // these 3 functions are run when certain buttons are clicked

    // changes difficulty property and conveys the change to the server

    change\_difficulty(new\_difficulty) {

        this.difficulty = new\_difficulty

        external\_change\_difficulty(new\_difficulty);

    }

    // simply disables the game, the only option now it to restart

    concede\_game() {

        this.just\_conceded = true

        this.user\_input\_disabled = true;

        this.over = true;

        this.highlighted\_squares = []

        this.selected\_from\_vector = null

        this.update\_board\_widget()

    }

    // this function resets the board properties to the initial game data and then informs the server of this change

    reset\_game() {

        let temp\_difficulty = this.difficulty;

        this.update\_from\_server\_data(INITIAL\_GAME\_DATA);

        this.difficulty = temp\_difficulty;

        this.just\_reset = true

        this.just\_conceded = false

        this.update\_board\_widget()

        external\_reset\_game();

    }

    // this function handles implementing a user move

    // it validates that the user can go

    // it then updates board's properties with the new board data from the server

    async make\_user\_move(from\_vector, to\_vector) {

        assert(

            this.next\_to\_go == "W",

            "User must be next to go in order to implement a user move"

        )

        // console.log("make\_user\_move called")

        let server\_data = await external\_implement\_user\_move\_and\_update(from\_vector, to\_vector);

        // console.log("updating the board with this server data")

        // console.log({server\_data})

        this.update\_from\_server\_data(server\_data, true)

    }

    // this function implements a computer move

    async make\_computer\_move() {

        console.log("make\_computer\_move called")

        // validates that it is the computer's turn

        assert(

            this.next\_to\_go == "B",

            "User must be next to go in order to implement a user move"

        )

        // created a promise to wait some amount of time (1 sec)

        const wait\_before\_displaying\_promise = new Promise((resolve) => setTimeout(resolve, 1000))

        // const wait\_a\_second\_promise = new Promise((resolve) => setTimeout(resolve, 0))

        // gets the new server data after computer move

        let server\_data = await external\_implement\_computer\_move\_and\_update();

        // console.log({server\_data})

        // get and remove move special data

        let move\_description = server\_data["computer\_move\_description"];

        delete server\_data["computer\_move\_description"];

        // console.log({ move\_description })

        // move description unpacked

        let [from\_vector, movement\_vector] = move\_description.move

        let resultant\_vector = add\_vectors(from\_vector, movement\_vector)

        // no change made for a second to the user can see the board after their move

        await wait\_before\_displaying\_promise

        // of the game is reset of conceded in that time, abort the function

        if (this.just\_reset || this.just\_conceded) {

            return null

        }

        // the relevant highlighting is applied to show the computer's move

        this.selected\_from\_vector = from\_vector

        this.possible\_to\_vectors = [resultant\_vector,]

        this.update\_board\_widget()

        // the move is highlighted for 0.8 seconds before the new board state is shown

        const wait\_to\_show\_move\_highlights = new Promise((resolve) => setTimeout(resolve, 800));

        await wait\_to\_show\_move\_highlights

        // again, abort function if game reset / conceded in that time

        if (this.just\_reset || this.just\_conceded) {

            return null

        }

        // console.log(`Implementing computer move:`)

        // console.log(JSON.stringify(move\_description))

        // console.log(`Updating board to:`)

        // console.log(JSON.stringify(server\_data.pieces\_matrix))

        // show the new board positions after the computer's move

        this.update\_from\_server\_data(server\_data, true);

        // console.log("move\_description.taken\_piece")

        // console.log(move\_description.taken\_piece)

        // if (arrays\_are\_equal(move\_description.taken\_piece, [null, null])) {

        //     alert(`Computer moved ${move\_description.moved\_piece[1]} from ${move\_description.from\_square} to ${move\_description.to\_square}`)

        // }

        // else {

        //     alert(`Computer moved ${move\_description.moved\_piece[1]} from ${move\_description.from\_square} to ${move\_description.to\_square} taking ${move\_description.taken\_piece}`)

        // }

    }

    // this function choreographs implementing the user move and then the computer move

    // if checks if the game is over after each one

    async user\_move\_and\_computer\_move\_cycle(from\_vector, to\_vector) {

        await this.make\_user\_move(from\_vector, to\_vector)

        if (this.game\_over) {

            this.handle\_game\_over()

            return null

        }

        // if (this.check) {

        //     alert("CHECK");

        // }

        await this.make\_computer\_move()

        if (this.game\_over) {

            this.handle\_game\_over()

            return null

        }

        // if (this.check) {

        //     alert("CHECK");

        // }

        // the user can now access the board to input another move

        this.user\_input\_disabled = false

    }

    // this function displays the appropriate output when the game is over

    handle\_game\_over() {

        // make sure the game is over

        assert(this.game\_over)

        let winning\_player = this.over\_data.winning\_player;

        let victory\_classification = this.over\_data.victory\_classification;

        // manually set the title message to show this

        let msg;

        switch (victory\_classification) {

            case "checkmate":

                msg = (winning\_player == 1) ? "Checkmate: congratulations, you won!" : "Checkmate: you lost, better luck next time"

                break

            case "stalemate board repeat":

                msg = "Stalemate: Threefold Repetition (Draw)"

                break

            case "stalemate no legal moves":

                msg = "Stalemate (Draw)"

                break

            default:

                throw new error("Invalid victory classification")

        }

        // update the title message

        this.update\_main\_title\_widget(msg);

        // the board remains disabled

        // alert(msg)

        // this.reset\_game();

    }

    // this handler function responds to the user clicking a certain square

    handle\_square\_click(vector) {

        // if user input is disabled, ignore the click

        if (this.user\_input\_disabled) {

            // alert("board disabled")

            // console.log("board disabled")

            return null;

        }

        // else disable the board and decide what to do

        this.user\_input\_disabled = true;

        // check if they have clicked a green highlighted square they could move to,

        // let from\_square\_already\_selected = this.selected\_from\_vector !== null;

        let is\_valid\_move = two\_d\_array\_contains\_sub\_array(this.possible\_to\_vectors, vector);

        // console.log(`Checking if  [${vector}]  in  ${JSON.stringify(this.possible\_to\_vectors)}  -->  result was  ${is\_valid\_move}`);

        // if they have, cause the appropriate user then computer move

        if (is\_valid\_move) {

            // async function call

            // console.log("about to call make\_user\_move")

            // this.make\_user\_move(this.selected\_from\_vector, vector);

            this.user\_move\_and\_computer\_move\_cycle(this.selected\_from\_vector, vector);

        }

        // else if not valid or no piece already selected then reselect

        else {

            // get piece at vector clicked

            let [i, j] = vector;

            let [row, col] = [7 - j, i];

            let [color, \_] = this.pieces\_matrix[row][col];

            // if the piece is one of the users then highlight red

            if (color == this.next\_to\_go) {

                this.selected\_from\_vector = vector;

            }

            else {

                this.selected\_from\_vector = null;

            }

            // then iterate through the legal moves and identify any squares that the user could move the selected piece to

            // if there are any, highlight green

            // if (color == this.next\_to\_go) {

            if (color == "W") {

                let position\_vector; let movement\_vector; let resultant\_vector;

                this.possible\_to\_vectors = [];

                for (let i in this.legal\_moves) {

                    [position\_vector, movement\_vector] = this.legal\_moves[i];

                    if (arrays\_are\_equal(position\_vector, vector)) {

                        resultant\_vector = add\_vectors(position\_vector, movement\_vector);

                        this.possible\_to\_vectors.push(resultant\_vector);

                    }

                }

            }

            // update the board to show any highlighting changes

            this.update\_board\_widget()

        }

        this.user\_input\_disabled = false

    }

    // this method returns an array of position vectors of squares and there color

    // it highlights the selected from vector red and all selected to vectors green

    get\_highlighted\_squares() {

        // returns a 2d array of vector and then color

        let highlighted\_squares = [];

        if (this.selected\_from\_vector !== null) {

            highlighted\_squares.push([

                this.selected\_from\_vector,

                "red"

            ]);

            for (let i in this.possible\_to\_vectors) {

                highlighted\_squares.push([

                    this.possible\_to\_vectors[i],

                    "green"

                ]);

            }

        }

        // console.log({highlighted\_squares})

        return highlighted\_squares;

    }

}

The main functions of this file are:

* Define 5 asynchronous functions that can be used to communicate with the server. To do this I created a general promised factory to ensure that I kept repeat logic to a minimum
* Define the Chess\_Board class.
  + This will contain data about the current chess board state in public properties. These properties will be accessed by the DOM manipulation methods.
  + It will contain handler functions for the 4 main inputs, square click, reset or concede button click and difficulty radio button change
  + It uses internal variables to keep track of the 2 square clicks needed to input a move and to produce the relevant highlighting in the process
  + It will be able to orchestrate the execution of a user’s move and then the computer move. This is in addition to checking if the game is over, and if so producing the correct output, after each move

I also used a small file called vector.js that contained some logic around adding vectors:

// the following functions execute add or subtract operations on a pair of vectors

function add\_vectors(v1, v2) {

    // v1 + v2

    return [

        v1[0] + v2[0],

        v1[1] + v2[1],

    ]

}

function subtract\_vectors(v1, v2) {

    // v1 - v2

    return [

        v1[0] - v2[0],

        v1[1] - v2[1],

    ]

}

// use arrays are equal method instead

// function vectors\_are\_equal(v1, v2) {

//     return JSON.stringify(v1.map(Number)) == JSON.stringify(v2.map(Number))

// }

The only remaining component of the frontend was the CSS:

#board {

    width: 80vmin;

    height: 80vmin;

    /\* width: 800px;

    height: 800px; \*/

    /\* margin-top: 10vmin;

    margin-bottom: 5vmin; \*/

    /\* margin-top: 0;

    margin-bottom: 0; \*/

    /\* margin-top: 2vmin;

    margin-bottom: 2vmin; \*/

    margin-left: auto;

    margin-right: auto;

    display: flex;

    flex-wrap: wrap;

    border: 2px solid black;

    border-collapse: collapse;

}

.square {

    width: 10vmin;

    height: 10vmin;

    /\* width: 100px;

    height: 100px; \*/

    display: flex;

    align-items: center;

    justify-content: center;

    font-size: 9vmin;

    /\* text-shadow: -1px -1px 0 #000, 1px -1px 0 #000, -1px 1px 0 #000, 1px 1px 0 #000; \*/

}

#top\_title {

    /\* height: 8vh; \*/

    height: 6vmin;

    width: 80vw;

    font-size: 6vmin;

    margin: 0 auto;

    text-align: center;

    /\* border: 2px solid red; \*/

    padding-top: 0;

    margin-top: 0;

    border-top: 0;

      /\* clip-path: polygon(0 0, 100% 0, 100% 30%, 0 100%); \*/

}

body {

    width: 100%;

    height: 100%;

    margin: 0 auto;

    text-align: center;

}

/\* .button\_wrapper { \*/

.centered {

    display: flex;

    align-items: center;

    justify-content: center;

    flex-direction: column;

}

.option\_button\_container {

    display: flex;

    justify-content: space-between;

    /\* width: 100vmin; \*/

    width: 60vmin;

    /\* margin-top: 2vh; \*/

}

.option\_button {

    /\* height: 6vmin;

    width: 8vmin;

    min-height: 4.5ch;

    min-width: 8ch;

    font-size: 5vmin; \*/

    /\* height: 6vmin;

    width: 8vmin;

    min-width: 14ch;

    font-size: 5vmin; \*/

    height: 6vmin;

    width: 8vmin;

    min-width: 8ch;

    font-size: 4vmin;

    text-align: center;

    /\* font-size: 5vmin; \*/

    /\* margin-top: 2vh; \*/

    /\* border: 2px  solid green; \*/

    /\* margin-left: 5vw;

    margin-right: 5vw; \*/

}

.option\_button:first-child {

    margin-left: 0;

    /\* margin-right: 2ch; \*/

}

.option\_button:last-child {

    /\* margin-left: 2ch; \*/

    margin-right: 0;

}

.pieces\_taken\_table th, td {

    width: 35vw;

    border: 2px solid black;

    font-size: 28px;

}

.spacer {

    display: block;

    /\* height: 5vh; \*/

    /\* height: 3vmin; \*/

    height: 2.5vmin;

    width: 100%;

    margin: 0;

    padding: 0;

    border: none;

    overflow: hidden;

    font-size: 0;

    line-height: 0;

}

/\*

.small\_text {

    font-size: 4vmin;

} \*/

#difficulty\_form form {

    display: inline-flex;

    flex-direction: row;

}

.difficulty\_form div{

    /\* width: 80vh; \*/

    width: 80vmin;

    /\* width: 100vmin; \*/

    /\* background-color: orange; \*/

    /\* border: 2px solid orange; \*/

}

.difficulty\_form label {

    margin-right: 6vw;

    width: 10vh;

    height: 5vh;

    /\* font-size: 4vh; \*/

    font-size: 4vmin;

    /\* margin: 5vh, auto; \*/

}

h1 {

    font-size: 4vmin;

    padding-top: 0;

    padding-bottom: 0;

    margin-top: 0;

    margin-bottom: 0;

    border-top: 0;

    border-bottom: 0;

}

#which\_pieces\_taken\_white {

    background-color: #66443a;

    font-size: 4vmin;

    color: white;

    text-shadow: -2px -2px 0 #000,

        2px -2px 0 #000,

        -2px 2px 0 #000,

        2px 2px 0 #000;

}

#which\_pieces\_taken\_black {

    background-color: #fae09f;

    font-size: 4vmin;

    color: black;

    /\* text-shadow: -1px -1px 0 #fff,

        1px -1px 0 #fff,

        -1px 1px 0 #fff,

        1px 1px 0 #fff; \*/

}

/\* body {background-color: red} \*/

It took a lot of experimentation but the resulting CSS is able to dictate the layout of each widget on the board, the widgets relative dimensions and its aesthetics such as color and text size.

The other design objective that the CSS archives is ensuring that the webpage woks on a range of aspect ratios.

This was done through the use of CSS properties like:

* Vmin: percentage of height or width, whichever is smaller
* Vmax: percentage of height or width, whichever is larger
* Vh: percentage of view height
* Vw: percentage of view width.

Here is the final result:

Table

Description automatically generated

Timeline

Description automatically generated

The above 2 screenshots show how the page looks from a landscape orientation with a laptop.

I will now show some other aspect ratios using chrome dev tools:

Graphical user interface, application, table

Description automatically generated Graphical user interface, application

Description automatically generated

Above is the webpage viewed in a landscap oerentation with the ipad air.

Graphical user interface

Description automatically generated with medium confidence Table

Description automatically generated

Above you can see the program running in a prtrait iphone 8. As can be seen, some of the elements are overly large. For instance the font size of the pieces taken and moves history tables are very large. However, all the elemets are visible and no are to large or small to see of interact with. This means that even though the sytles do not perfectly translate to every device, the user interface is still perfectly functional and just as usable on a wide range of devices and aspect ratios.

Here are some screenshot of the game as I play the computer:

I have opened a new game on medium difficulty:

A picture containing square

Description automatically generated

We can see that the computer has prompted me that it is my go. I will not click the white pawn in front of the queen:

A picture containing chart

Description automatically generated

If I click off this square onto another square I get back to the blank board I started with. If I click some of my other pieces they are also highlighted red. If they can make any moves, these are highlighted in green.

A picture containing square

Description automatically generated

A picture containing square

Description automatically generated

A picture containing table

Description automatically generated

Clicking other squares including the computers squares remove the highlighting but doesn’t cause any unexpected behaviours (such as moving for the computer or moving my piece to that square).

I will click the front most green square. I have set the difficulty to be higher to slow the program down and allow me to easily capture what happens when the computer moves. Changing the difficulty has not affected the highlighting.

And here is a screenshot of the below widgets before I move:

Graphical user interface, application

Description automatically generated

Then I clicked the front most green square and saw:

That my pawn has moved forwards, the title element had changed and the move history had changed.

A picture containing table

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

I then saw that after my move, the computer made its move. This was show by the following highlighting for 0.8 seconds

A picture containing table

Description automatically generated

Then the board updated again to show the computer’s move and the main title told me it was my turn:

A picture containing treemap chart

Description automatically generated

The move history widget had updated:

Graphical user interface

Description automatically generated with medium confidence

I was able to click to move another piece, one of my knights forward:

Treemap chart

Description automatically generated with medium confidence A picture containing treemap chart

Description automatically generated

Graphical user interface, timeline

Description automatically generated with medium confidence

Then the computer moved again:

A picture containing treemap chart

Description automatically generated

A picture containing graphical user interface

Description automatically generated

A picture containing table

Description automatically generated

I was then able to close the tab and reopen it to the exact same game.

I saw this notice:

Treemap chart

Description automatically generated

Then the tab closed and I reopened it on the same chess game. It realise the message may be a little misleading but I am not sure how to change it.

I wat then able to click the concede button.

A picture containing square

Description automatically generated

This caused the main message at the top to update and the board to become non-interactable.

I was able to change the game’s difficulty an then press restart to revert the game to the starting board positions and a blank move history:

A picture containing table

Description automatically generated

Timeline

Description automatically generated

The new difficulty I set before restarting the game was preserved

Here is link to a MP4 video file on google drive where I show the chess game and its features:

Link here

Testing:

To test the program I first added to my automated tests to add functionality to test some of the additional modules and features created. I then tested the program by running the website and playing chess against it. By testing the program as if I were a user, using the program I was able to identify various issues. I then created unit tests to identify some of the issues and then fixed my code to prevent the issue. I was then able to run the unit test again to verify that the issue had been solved.

Firstly I created additional minimax tests and combined together all minimax tests into one test file:

# this test is responsible for testing various mutations of the minimax function and how they play, it is not a data driven test

# imports of external modules.

from random import choice as random\_choice, randint

# from itertools import product as iter\_product

import pickle

import unittest

# from functools import wraps

import multiprocessing

import os

import csv

from time import perf\_counter

# imports of local modules

from chess\_game import Game

# from chess\_functions import Board\_State

# from minimax import Move\_Engine

from move\_engine import Move\_Engine\_Timed, Move\_Engine\_Prime

from chess\_functions import random\_board\_state

# from assorted import ARBITRARILY\_LARGE\_VALUE

# from board\_state import Board\_State

# from vector import Vector

# # was not needed in the end, this decorator would have repeated a given function a given number of times

# def repeat\_decorator\_factory(times):

#     def decorator(func):

#         @wraps(func)

#         def wrapper(\*args, \*\*kwargs):

#             for \_ in range(times):

#                 func(\*args, \*\*kwargs)

#         return wrapper

#     return decorator

# this is a utility function that maps a function across an iterable but also converts the result to a list data structure

def list\_map(func, iter):

    return list(map(func, iter))

# this function is used to serialize a pieces matrix for output in a message

# it converts pieces to symbols it they are not none

def map\_pieces\_matrix\_to\_symbols(pieces\_matrix):

    return list\_map(

        lambda row: list\_map(

            lambda square: square.symbol() if square else None,

            row

        ),

        pieces\_matrix

    )

# this functions updates a CSV file with the moves and scores of a chess game for graphical analysis in excel

# each move in the game causes a new row to be added

def csv\_write\_move\_score(file\_path,  move\_counter, move, score):

    # convert move to a pair of squares

    position\_vector, movement\_vector = move

    resultant\_vector = position\_vector + movement\_vector

    from\_square = position\_vector.to\_square()

    to\_square = resultant\_vector.to\_square()

    # # if file doesn't exist, create is and add the headers

    # if not os.path.exists(file\_path):

    #     with open(file\_path, "w", newline="") as file:

    #         writer = csv.writer(file, delimiter=",")

    #         writer.writerow(("from\_square", "to\_square", "score"))

    # add data as a new row

    with open(file\_path, "a", newline="") as file:

        writer = csv.writer(file, delimiter=",")

        # writer.writerow(("Move\_counter", "B\_score", "from\_square", "to\_square", "W\_score"))

        writer.writerow((move\_counter, -score, from\_square, to\_square, score))

# this function is used to time another function to allow for performance testing

def time\_function(function):

    start = perf\_counter()

    result = function()

    end = perf\_counter()

    time\_delta = end - start

    return result, time\_delta

# this function aims to save games to a file as part of an unfinished feature

def save\_games(games: tuple, file\_path):

    with open(file\_path, "wb") as file:

        file.write(

            pickle.dumps(

                games

            )

        )

# this method aims to load a game from a file as part of another unfinished feature

def load\_games(file\_path):

    if not os.path.exists(file\_path):

        return dict()

    with open(file\_path, "rb") as file:

        return pickle.loads(

            file.read()

        )

# the below function contains the logic to perform a test between 2 minimax bots to assert that the good bot is better

# this contains the majority of the logic to do a bot vs bot test with the game class

# it is a component as it isn't the whole test

def minimax\_test\_component(description, good\_bot, bad\_bot, success\_criteria, write\_to\_csv, csv\_folder, csv\_file\_name, load\_game=False, save\_game=False):

    # print(f"CALL minimax\_test\_component(description={description}, write\_to\_csv={write\_to\_csv})")

    # sourcery skip: extract-duplicate-method

    # good bot and bad bot make decisions about moves,

    # the test is designed to assert that good bot wins (and or draws in some cases)

    # if the write to csv file it enabled

    # generate csv path

    if write\_to\_csv:

        # not sure why but the description sometimes contains an erroneous colon, this is caught and removed

        # was able to locate bug to here, as it is a test I added a quick fix

        # bug located, some description stings included them

        # create folders

        if not os.path.exists(csv\_folder):

            os.makedirs(csv\_folder)

        # create file with headers if is doesn't exist

        csv\_path = f"{csv\_folder}/{csv\_file\_name}.csv"

        # overwrite so it is blank

        with open(csv\_path, "w", newline="") as file:

            writer = csv.writer(file, delimiter=",")

            writer.writerow(("Move\_counter", "B\_score", "from\_square", "to\_square", "W\_score"))

    # pickle\_file\_path = f"{csv\_folder}/{csv\_file\_name}.games"

    # if load\_game and os.path.exists(pickle\_file\_path):

    #     print("Attempting to load game")

    #     indexes = (-1, -2)

    #     games = load\_games(pickle\_file\_path)

    #     for game in games:

    #         print(hash(game))

    #         print(game.board\_state.next\_to\_go)

    #     for index in indexes:

    #         try:

    #             game = games[index]

    #             assert game.board\_state.next\_to\_go == "W", "next to go wasn't user so use other game"

    #         except Exception as e:

    #             print(e)

    #             continue

    #         else:

    #             print(f"Using game {hash(game)} at index {index}")

    #             games = games[:index]

    # else:

    # create a new game

    print("Creating new game")

    game: Game = Game(echo=True)

        # games = tuple()

    # start a new blank game

    # depth irrelevant as computer move function passed as parameter

    print()

    print(f"Beginning test game:   {description}")

    # def record\_new\_game(games, new\_game):

    #     games = tuple(list(games) + [new\_game])

    #     save\_games(games, pickle\_file\_path)

    #     return games

    # if save\_game:

    #     games = record\_new\_game(games, game)

    # keep them making moves until return statement breaks loop

    while True:

        # print()

        # game.board\_state.print\_board()

        # print()

        # get move choice from bad bot

        # the bad bots move is implemented as the user as the user plays as white (advantage)

        result, time\_delta = time\_function(

            lambda: bad\_bot(game)

        )

        score, move\_choice = result

        # print({"move\_choice": move\_choice})

        # serialised to is can be passed as a user move (reusing game class)

        position\_vector, movement\_vector = move\_choice

        resultant\_vector = position\_vector + movement\_vector

        from\_square, to\_square = position\_vector.to\_square(), resultant\_vector.to\_square()

        # implement bad bot move and update csv

        # move, score = game.implement\_user\_move(from\_square=from\_square, to\_square=to\_square, time\_delta=time\_delta)

        move, \_ = game.implement\_user\_move(from\_square=from\_square, to\_square=to\_square, time\_delta=time\_delta, estimated\_utility=score)

        # games = record\_new\_game(games, game)

        if write\_to\_csv:

            csv\_write\_move\_score(

                file\_path=csv\_path,

                move=move,

                score=game.board\_state.static\_evaluation(),

                move\_counter=game.move\_counter

            )

        # piece\_moved = game.board\_state.get\_piece\_at\_vector(resultant\_vector)

        # print(f"Move {game.move\_counter}: bad bot moved {piece\_moved} from {from\_square} to {to\_square} with a score perceived of {score}")

        # if game.board\_state.color\_in\_check():

        #    print(f"CHECK:  {game.board\_state.next\_to\_go} in check")

        # see if this move causes the test to succeed or fail or keep going

        # use the provided success criteria to determine if the game should be over.

        success, msg, board\_state = success\_criteria(game, description=description)

        # result = success\_criteria(game, description=description)

        # print({"result": result})

        # success, msg, board\_state = result

        if success is not None:

            # game.board\_state.print\_board()

            return success, msg, board\_state

        # repeat for the good bot

        # providing good bot function, implement good bot move and update csv

        move, score = game.implement\_computer\_move(best\_move\_function=good\_bot)

        # games = record\_new\_game(games, game)

        # update CSV

        if write\_to\_csv:

            csv\_write\_move\_score(

                file\_path=csv\_path,

                move=move,

                score=game.board\_state.static\_evaluation(),

                move\_counter=game.move\_counter

            )

        # unpack move in terms of to and from squares

        position\_vector, movement\_vector = move

        resultant\_vector = position\_vector + movement\_vector

        # piece\_moved = game.board\_state.get\_piece\_at\_vector(resultant\_vector)

        from\_square, to\_square = position\_vector.to\_square(), resultant\_vector.to\_square()

        # print(f"Move {game.move\_counter}: good bot moved {piece\_moved} from {from\_square} to {to\_square} with a score perceived of {score}")

        # if game.board\_state.color\_in\_check():

        #    print(f"CHECK:  {game.board\_state.next\_to\_go} in check")

        # again check if this affects the test

        success, msg, board\_state = success\_criteria(game, description=description)

        if success is not None:

            # game.board\_state.print\_board()

            return success, msg, board\_state

        # # if needed provide console output to clarify that slow bot hasn't crashed

        # if game.move\_counter % 10 == 0 or depth >= 3:

        # print(f"Moves {game.move\_counter}:  static evaluation -> {game.board\_state.static\_evaluation()}, Minimax evaluation -> {score} by turn {description}")

# below are some function that have been programmed as classes with a \_\_call\_\_ method.

# these are basically fancy functions that CAN BE HASHED.

# I had to manually do this under the hood hashing as it is needed to allow communication between the threads

# a job must be hashable to be piped to a thread (separate python instance)

# this is a pipe-able object that makes a random move

class Random\_Bot():

    # picks a random move

    def \_\_call\_\_(self, game):

        # determine move at random

        legal\_moves = list(game.board\_state.generate\_legal\_moves())

        assert len(legal\_moves) != 0

        # match minimax output structure

        # score, best\_move

        return None, random\_choice(legal\_moves)

    def \_\_hash\_\_(self) -> int:

        return hash("I am random bot, I am a unique singleton so each instance can share a hash")

# this bot picks a good move

# it exploration is limited by time

# has constructor to allow for configuration

class Bot\_By\_Time():

    # configure for depth and allow variable depth

    def \_\_init\_\_(self, time, cache\_allowed=False):

        self.time = time

        self.move\_engine = Move\_Engine\_Timed()

        self.move\_engine.cache\_allowed = cache\_allowed

    # make minimax function call given config

    def \_\_call\_\_(self, game):

        return self.move\_engine(

            board\_state=game.board\_state,

            time=self.time

        )

        # result = self.move\_engine(

        #     board\_state=game.board\_state,

        #     time=self.time

        # )

        # print(f"Bot\_By\_Time:   result={result}")

        # return result

    def \_\_hash\_\_(self) -> int:

        return hash(f"Bot\_By\_Time(time={self.time})")

# this is a bot that has its exploration limited by depth

class Bot\_By\_Depth():

    # configure for depth and allow variable depth

    def \_\_init\_\_(self, depth, cache\_allowed=False):

        self.depth = depth

        self.move\_engine = Move\_Engine\_Prime()

        self.move\_engine.cache\_allowed = cache\_allowed

    # make minimax function call given config

    def \_\_call\_\_(self, game):

        return self.move\_engine(

            board\_state=game.board\_state,

            depth=self.depth

        )

        # result = self.move\_engine(

        #     board\_state=game.board\_state,

        #     depth=self.depth

        # )

        # print(f"Bot\_By\_Depth:   result={result}")

        # return result

    def \_\_hash\_\_(self) -> int:

        return hash(f"Bot\_By\_Depth(depth={self.depth})")

# used to look at a game and decide if the test should finish

class Success\_Criteria():

    # constructor allow config for stalemates to sill allow test to pass

    def \_\_init\_\_(self, allow\_stalemate\_3\_states\_repeated: bool):

        self.allow\_stalemate\_3\_states\_repeated = allow\_stalemate\_3\_states\_repeated

    def \_\_call\_\_(self, game: Game, description):

        # returns: success, message, serialised pieces matrix

        # call game over and use a switch case to decide what to do

        match game.check\_game\_over():

            # if 3 repeat stalemate, check with config wether is is allows

            # case True, None, "Stalemate":

            case True, None, \_:

                # game.board\_state.print\_board()

                if game.board\_state.is\_3\_board\_repeats\_in\_game\_history() and self.allow\_stalemate\_3\_states\_repeated:

                    # game.board\_state.print\_board()

                    # print(f"Success: Stalemate at {game.move\_counter} moves in test {description}: 3 repeat board states, outcome specify included in allowed outcomes")

                    # return True, f"Success: Stalemate at {game.move\_counter} moves in test {description}: 3 repeat board states, outcome specify included in allowed outcomes", map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix)

                    return (

                        True,

                        f"Success: Stalemate at in test {description}: 3 repeat board states, outcome specify included in allowed outcomes",

                        map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix),

                    )

                else:

                    # game.board\_state.print\_board()

                    # print(f"FAILURE: ({description}) stalemate caused (3 repeats?  -> {game.board\_state.is\_3\_board\_repeats\_in\_game\_history()})")

                    return (

                        False,

                        f"FAILURE: ({description}) stalemate caused (3 repeats?  -> {game.board\_state.is\_3\_board\_repeats\_in\_game\_history()})",

                        map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix),

                    )

            # good bot loss causes test to fail

            # case True, 1, "Checkmate":

            case True, 1, \_:

                # game.board\_state.print\_board()

                # print(f"Failure: ({description}) computer lost")

                return (

                    False,

                    f"Failure: ({description}) computer lost",

                    map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix),

                )

            # good bot win causes test to pass

            # case True, -1, "Checkmate":

            case True, -1, \_:

                # game.board\_state.print\_board()

                # print(f"SUCCESS: ({description}) Game has finished and been won in {game.move\_counter} moves")

                return (

                    True,

                    f"SUCCESS: ({description}) Game has finished and been won in {game.move\_counter} moves",

                    map\_pieces\_matrix\_to\_symbols(game.board\_state.pieces\_matrix),

                )

            # if the game isn't over, return success as none and test will continue

            case False, \_, \_:

                return (

                    None,

                    None,

                    None,

                )

    def hash(self):

        return hash(f"Success\_Criteria(allow\_stalemate\_3\_states\_repeated={self.allow\_stalemate\_3\_states\_repeated})")

# given a test package (config for one test), carry it out

def execute\_test\_job(test\_data\_package):

    # deal with unexplained bug where argument is tuple / list length 1 containing relevant dict (quick fix as only a test)

    # I was able to identify that this is where it occurs and add a correction but I am not sure what the cause of the bug is

    if any(isinstance(test\_data\_package, some\_type) for some\_type in (tuple, list)):

        if len(test\_data\_package) == 1:

            test\_data\_package = test\_data\_package[0]

    # print(f"test\_data\_package   -->   {test\_data\_package}")

    # really simple, call minimax test component providing all keys in package as keyword arguments

    return minimax\_test\_component(\*\*test\_data\_package)

# this pool jobs function for completing tests in parallel is not needed when the move engine is parallelized

# this function takes an iterable of hashable test\_packages

# it all 8 logical cores on my computer to multitask to finish the test sooner

def pool\_jobs(test\_data):

    # counts logical cores

    # my CPU is a 10th gen i7

    # it has 4 cores and 8 logical cores due to hyper threading

    # with 4-8 workers I can use 100% of my CPU

    cores = multiprocessing.cpu\_count()

    # create a pool

    with multiprocessing.Pool(cores) as pool:

        # map the execute\_test\_job function across the set of test packages using multitasking

        # return the result

        # return pool.map(

        #     func = execute\_test\_job,

        #     iterable = test\_data

        # )

        yield from pool.map(

            func=execute\_test\_job,

            iterable=test\_data

        )

# test case contains unit tests

# multitasking only occurs within a test, tests are themselves executed sequentially

# I could pool all tests into one test function but this way multiple failures can occur in different tests

# (one single test would stop at first failure)

overnight = False

RANDOM\_TRIAL\_NUM = 3

# this function asserts that are elements in a list are the same

def test\_same(some\_list):

    assert len(some\_list) > 0

    return all(some\_list[0] == element for element in some\_list)

# @unittest.skip("Takes too long")

class Test\_Case(unittest.TestCase):

    # this function takes the results of the tests from the test pool and checks the results with a unittest

    # a failure is correctly identified to correspond to the function that called this function

    # reduces repeated logic

    def check\_test\_results(self, test\_results):

        # for success, msg, final\_pieces\_matrix in test\_results:

        for success, msg, \_ in test\_results:

            # print()

            # for row in final\_pieces\_matrix:

            #     row = "  ".join(map(

            #         lambda square: str(square).replace("None", ". "),

            #         row

            #     ))

            #     print(row)

            # print(msg)

            # i choose to iterate rather than assert all as this allows me to have the appropriate message on failure

            self.assertTrue(

                success,

                msg=msg

            )

    # this function tests that a cached call always produces the same result as one without caching

    # it also ensures that the second cache call for the same search is much faster

    # this test is completed on many random board states

    def test\_cache\_is\_faster(self):

        trials = 40

        # trials = 5

        # create and configure move engines

        move\_engine\_with\_cache = Move\_Engine\_Prime()

        move\_engine\_without\_cache = Move\_Engine\_Prime()

        move\_engine\_without\_cache.cache\_allowed = False

        for move\_engine in (move\_engine\_with\_cache, move\_engine\_without\_cache):

            move\_engine.depth = 2

            move\_engine.cache\_manager.allow\_greater\_depth = False

        # repeat test for many trials

        for \_ in range(trials):

            # note random\_board\_state as a function is inefficient as it is used for testing only, takes a while

            # generate a random board state

            board\_state = random\_board\_state(60)

            # turn of 3 repeat stalemates so cache and without behave the same

            board\_state.three\_repeat\_stalemates\_enabled = False

            # board\_state.print\_board()

            # check result with out cache

            true\_result = move\_engine\_without\_cache(board\_state)

            true\_result = list(true\_result)

            true\_result[1] = list(true\_result[1])

            # attempt 1 with cache

            first\_cache\_result, first\_cache\_time\_delta = time\_function(

                lambda: move\_engine\_with\_cache(board\_state)

            )

            first\_cache\_result = list(first\_cache\_result)

            first\_cache\_result[1] = list(first\_cache\_result[1])

            # check that the results were the same

            self.assertEqual(

                first\_cache\_result,

                true\_result,

                "The first cache call should give the same result as without cache"

            )

            # attempt 2 with cache

            second\_cache\_result, second\_cache\_time\_delta = time\_function(

                lambda: move\_engine\_with\_cache(board\_state)

            )

            second\_cache\_result = list(second\_cache\_result)

            second\_cache\_result[1] = list(second\_cache\_result[1])

            # check that the results were again the same

            self.assertEqual(

                first\_cache\_result,

                true\_result,

                "The second cache call should give the same result as without cache"

            )

            # check that with cache was much faster

            self.assertLessEqual(

                second\_cache\_time\_delta,

                0.2 \* first\_cache\_time\_delta,

                "When using database cache, the lookup should be at least 5 times quicker than calculation"

            )

    # tests basic minimax vs random moves

    def test\_timed\_vs\_randotron(self):

        trials = RANDOM\_TRIAL\_NUM

        # test package generated to include relevant data and logic (bots and success criteria)

        def generate\_jobs():

            for num in range(1, trials+1):

                for time in [2, 5, 10, 20]:

                    yield {

                        "description": f"test: {time}s timed move engine vs randotron",

                        "good\_bot": Bot\_By\_Time(time=time),

                        "bad\_bot": Random\_Bot(),

                        "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

                        "write\_to\_csv": True,

                        "csv\_folder": "./test\_reports/test\_timed\_vs\_randotron",

                        "csv\_file\_name": f"test\_timed\_{time}s\_num\_{num}"

                    }

        def generate\_test\_results():

            for test\_specs in generate\_jobs():

                yield minimax\_test\_component(\*\*test\_specs)

        self.check\_test\_results(

            generate\_test\_results()

        )

    # tests basic minimax vs random moves

    @unittest.skip("takes too long")

    def test\_depth\_vs\_randotron(self):

            trials = RANDOM\_TRIAL\_NUM

            # test package generated to include relevant data and logic (bots and success criteria)

            def generate\_jobs():

                for num in range(1, trials+1):

                    # for depth in range(4):

                    for depth in [1, 2]:

                        yield {

                            "description": f"test: depth {depth} move engine vs randotron",

                            "good\_bot": Bot\_By\_Depth(depth=depth),

                            "bad\_bot": Random\_Bot(),

                            "success\_criteria": Success\_Criteria(

                                allow\_stalemate\_3\_states\_repeated=(depth<=1)

                            ),

                            "write\_to\_csv": True,

                            "csv\_folder": "./test\_reports/test\_depth\_vs\_randotron",

                            "csv\_file\_name": f"test\_depth\_{depth}\_num\_{num}"

                        }

            def generate\_test\_results():

                for test\_specs in generate\_jobs():

                    yield minimax\_test\_component(\*\*test\_specs)

            self.check\_test\_results(

                generate\_test\_results()

            )

    # test that bots that can explore more are better

    # @unittest.skip("takes too long")

    def test\_bots\_by\_depth(self):

        # test package generated to include relevant data and logic (bots and success criteria)

        def generate\_jobs():

            # don't do depth 0s as they cannot decide a move

            for depth\_greater in (3,):

            # for depth\_greater in (2, 3):

                # ensure depth\_a >= depth\_b

                # for depth\_lesser in range(1, depth\_greater):

                # for depth\_lesser in range(1, depth\_greater+1):

                # allow\_draw = (depth\_lesser == depth\_greater)

                depth\_lesser = depth\_greater - 1

                yield {

                    "description": f"test: depth {depth\_greater} bot vs depth {depth\_lesser} bot",

                    "good\_bot": Bot\_By\_Depth(depth=depth\_greater),

                    "bad\_bot": Bot\_By\_Depth(depth=depth\_lesser),

                    "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

                    # "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=allow\_draw),

                    "write\_to\_csv": True,

                    "csv\_folder": "./test\_reports/test\_bots\_by\_depth",

                    "csv\_file\_name": f"test\_depth\_{depth\_greater}\_vs\_depth\_{depth\_lesser}"

                }

        def generate\_test\_results():

            for test\_specs in generate\_jobs():

                yield minimax\_test\_component(\*\*test\_specs)

        self.check\_test\_results(

            generate\_test\_results()

        )

    # test that bots that can explore more are better

    def test\_bots\_by\_time(self):

        # test package generated to include relevant data and logic (bots and success criteria)

        def generate\_jobs():

            time\_deltas = [2, 5, 10, 15, 20]

            for time\_delta\_lesser in time\_deltas:

                time\_delta\_greater = max(1.4\*time\_delta\_lesser, 4+time\_delta\_lesser)

                # time\_delta\_greater = max(2\*time\_delta\_lesser, 10+time\_delta\_lesser)

                # time\_delta\_greater = max(4\*time\_delta\_lesser, 15+time\_delta\_lesser)

                yield {

                    "description": f"test: {time\_delta\_greater}s timed bot vs {time\_delta\_lesser}s timed bot",

                    "good\_bot": Bot\_By\_Time(time=time\_delta\_greater),

                    "bad\_bot": Bot\_By\_Time(time=time\_delta\_lesser),

                    # "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=False),

                    "success\_criteria": Success\_Criteria(allow\_stalemate\_3\_states\_repeated=True),

                    "write\_to\_csv": True,

                    "csv\_folder": "./test\_reports/test\_bots\_by\_time",

                    "csv\_file\_name": f"test\_{time\_delta\_greater}s\_timed\_bot\_vs\_{time\_delta\_lesser}s\_timed\_bot"

                }

        def generate\_test\_results():

            for test\_specs in generate\_jobs():

                yield minimax\_test\_component(\*\*test\_specs)

        self.check\_test\_results(

            generate\_test\_results()

        )

    # test what the parallel move engine doesn't produce a different output

    def test\_deterministic\_outcomes\_parallel(self):

        # create move engines and configures them

        with\_parallel\_engine = Move\_Engine\_Prime()

        without\_parallel\_engine = Move\_Engine\_Prime()

        for engine in (with\_parallel\_engine, without\_parallel\_engine):

            engine.cache\_allowed = False

            engine.additional\_depth = 0

        with\_parallel\_engine.parallel = True

        without\_parallel\_engine.parallel = False

        def always(\*args, \*\*kwargs): return True

        def never(\*args, \*\*kwargs): return False

        with\_parallel\_engine.should\_use\_parallel = always

        without\_parallel\_engine.should\_use\_parallel = never

        # generate a large number of random board states

        def gen\_random\_board\_states():

            for \_ in range(40):

                yield random\_board\_state(moves=randint(0, 10))

            for \_ in range(20):

                yield random\_board\_state(moves=randint(0, 20))

            for \_ in range(10):

                yield random\_board\_state(moves=randint(0, 40))

        # test that for each board state, the move engines produce the same deterministic outcome

        for board\_state in gen\_random\_board\_states():

            score\_parallel, move\_parallel = with\_parallel\_engine(board\_state)

            score\_non\_parallel, move\_non\_parallel = without\_parallel\_engine(board\_state)

            fail\_msg = f"Test hash={hash(board\_state)}: score parallel != without -->  {score\_parallel} != {score\_non\_parallel}"

            # assert score\_parallel == score\_non\_parallel, fail\_msg

            self.assertEqual(

                score\_parallel, score\_non\_parallel, fail\_msg

            )

            fail\_msg = f"Test hash={hash(board\_state)}: move parallel != without -->  {move\_parallel} != {move\_non\_parallel}"

            # assert move\_parallel == move\_non\_parallel, fail\_msg

            self.assertEqual(

                move\_parallel, move\_non\_parallel, fail\_msg

            )

        # for trial in range(1, 101):

        #     try:

        #         moves = random\_choice(range(10, 50))

        #         board\_state = random\_board\_state(moves)

        #         score\_parallel, \_ = with\_parallel\_engine(board\_state)

        #         score\_non\_parallel, \_ = without\_parallel\_engine(board\_state)

        #         fail\_msg = f"Test moves={moves} hash={hash(board\_state)}: score parallel != without -->  {score\_parallel} != {score\_non\_parallel}"

        #         assert score\_parallel == score\_non\_parallel, fail\_msg

        #     except AssertionError as e:

        #         print(f"TEST FAILURE: (trial={trial})")

        #         board\_state.print\_board()

        #         print(repr(board\_state))

        #         self.fail(str(e))

        #     except Exception as e:

        #         self.fail(f"UNEXPECTED ERROR (trial = {trial}):   {str(e)}")

if \_\_name\_\_ == '\_\_main\_\_':

    unittest.main()

I added tests to verify that the minimax algorithm was faster when if encountered a board state that was in the cache.

I also added a test to verify that the output produced by the new improved parallel minimax algorithm was the same when the completing the search in parallel and when completing the search without any concurrency.

The final test I added were to test the time limited minimax algorithm by checking that bots given move time to explore the tree did better than those given less time.

These tests allowed me to verify that my minimax algorithm was still working after I modified it. It was immediately apparent if the change caused the algorithm to crash of produce random moves.

I also created some new tests for the game class:

import unittest

import pickle

from .game import Game

from assorted import NotComputerTurn, NotUserTurn

from chess\_functions import King, Queen, Rook, Bishop, Pawn, Board\_State, Vector

from move\_engine import Move\_Engine\_Prime

class Test\_Case(unittest.TestCase):

    # this test checks that a game object can be preserved when it is encoded to binary and then loaded back

    def test\_save\_and\_restore(self):

        game = Game(

            echo=False,

            time=10,

        )

        # make a load of changes to the game object

        game.implement\_user\_move(from\_square="A2", to\_square="A4")

        game.implement\_computer\_move(best\_move\_function=lambda \*arg, \*\*kwargs: [0, [Vector(0, 6), Vector(0, -2)]])

        game.implement\_user\_move(from\_square="B2", to\_square="B4")

        game.implement\_computer\_move(best\_move\_function=lambda \*arg, \*\*kwargs: [0, [Vector(1, 6), Vector(0, -2)]])

        game.implement\_user\_move(from\_square="C2", to\_square="C4")

        game.implement\_computer\_move(best\_move\_function=lambda \*arg, \*\*kwargs: [0, [Vector(2, 6), Vector(0, -2)]])

        original\_game = game

        pickle\_file\_path = "chess\_game/test\_data/test\_save\_and\_restore/saved\_game.game"

        # dump it to binary and save to a file

        with open(pickle\_file\_path, "wb") as file:

            file.write(

                pickle.dumps(

                    game

                )

            )

        # read the binary and load it back into an object

        with open(pickle\_file\_path, "rb") as file:

            reloaded\_game = pickle.loads(

                file.read()

            )

        # check that the games are the same

        self.assertTrue(

            original\_game == reloaded\_game,

            "The original game was not the same as the saved and reloaded game"

        )

These were used to check that I could save a game object to binary and then restore it without its contents being changed.

    def test\_whose\_go(self):

        # this function tests that the validation present can stop a player going twice in a row

        game = Game(time=2)

        game.implement\_user\_move(from\_square="A2", to\_square="A4")

        def try\_to\_make\_user\_move(\*args, \*\*kwargs):

            game.implement\_user\_move(from\_square="B2", to\_square="B4")

        def try\_to\_make\_computer(\*args, \*\*kwargs):

            game.implement\_computer\_move()

        self.assertRaises(

            NotUserTurn,

            try\_to\_make\_user\_move,

            "The user shouldn't be able to make 2 consecutive turns"

        )

        game.implement\_computer\_move()

        self.assertRaises(

            NotComputerTurn,

            try\_to\_make\_computer,

            "The computer shouldn't be able to make 2 consecutive turns"

        )

I also created the above test to ensure the validation of the Game class was working and that it wouldn’t let the wrong player go. To do this I asserted that it raised the appropriate errors.

Here are some screenshots as evidence that the old tests and my new tests are able to run successfully:

Text

Description automatically generated

![A computer screen capture

Description automatically generated with medium confidence]()

The dots represent completed test. The first image show all 80 of the fast test have been completed. The second image shows the last test to finish form the slow minimax test (which I ran overnight). As the game output function has repeatedly cleared the terminal we cannot see the dots for the rest of the test. I could tell that the tests were finished as there was no python task in the task manager and as this test hadn’t been cleared away as another test was completed.

In the process of getting these test to run successfully I identified various issues due to initially failing test which were later fixed

I then tested the website and the program as a whole (black box testing) by playing games of chess against it. This helped me identify more issues that I was able to fix.

Here are the main issues that caused tests to fail and how I fixed them:

Timed Minimax not working:

From the automated testing of the timed minimax function I realised that the function was not consistently interrupting itself and sticking to the provided maximum time. This was because I not checking the time and interrupting the function frequently enough. When I completed a depth 3 call, I was only checking the time after pre-sorting the moves and after each depth 2 recursive call was completed. To fix this I added a max time argument to the main minimax method of the original Move Engine class.

def minimax(self, board\_state: Board\_State, depth, alpha=0-(ARBITRARILY\_LARGE\_VALUE+1), beta=ARBITRARILY\_LARGE\_VALUE+1, variable\_depth: bool | None = None, part\_of\_presort=False, legal\_moves\_to\_examine: Iterable | None = None, max\_time=None) -> tuple[int, tuple[Vector, Vector]]:

It had a default value of None which meant that there was no max time. This meant that previous code that called the minimax function without a max time parameter was unaffected. I then added a relevant function to interrupt the minimax function:

# this function is used to ensure that the minimax function can be self interrupting

def is\_time\_expired(max\_time):

    if max\_time is None:

        return False

    # print("🕰️", end="")

    return perf\_counter() >= max\_time

and regularly checked if the max time had been exceeded throughout the minimax function:

        # self interrupting

        if is\_time\_expired(max\_time):

           raise TimeOutError()

I also passed on this max time argument on to all recursive calls

                # recursive minimax call with depth n-1 to evaluate child

                score, \_ = self.minimax(

                    board\_state=child\_board\_state,

                    depth=depth-1,

                    alpha=alpha,

                    beta=beta,

                    # is\_time\_expired=is\_time\_expired,

                    max\_time=max\_time,

                )

This meant that the minimax timed call now more reliably stuck to its designated max time.

Here is a screen shot of a unit test to show that the issue was fixed:

A picture containing text, electronics

Description automatically generated

Calendar

Description automatically generated

The above screenshot were for a game where a 5 second bot played against a 9 second bot. The times column show that now, the bots are able to stick to their specified time with only a few small exceptions

Some test were written as I wrote the code or even before. This test driven development meant that I aimed to write the code to make the test work. For instance when I created the test to check if game objects could be pickled:

    def test\_save\_and\_restore(self):

it failed the first time. This prompted me to define a \_\_eq\_\_ method for the Game\_Webiste class which caused it to then work.

Pawns cannot move forward 2 places:

Other issues were encountered when I performed black box tests.

I identified that I couldn’t move more than one Pawn forward 2 spaces.

I identified the issue and was able to fix the code:

STARTING\_POSITIONS: tuple[tuple[pieces\_mod.Piece]] = (

    (

        pieces\_mod.Rook(color="B"),

        pieces\_mod.Knight(color="B"),

        pieces\_mod.Bishop(color="B"),

        pieces\_mod.Queen(color="B"),

        pieces\_mod.King(color="B"),

        pieces\_mod.Bishop(color="B"),

        pieces\_mod.Knight(color="B"),

        pieces\_mod.Rook(color="B")

    ),

    # (pieces\_mod.Pawn(color="B"),)\*8,

    tuple(pieces\_mod.Pawn(color="B") for \_ in range(8)),

    (None,)\*8,

    (None,)\*8,

    (None,)\*8,

    (None,)\*8,

    # (pieces\_mod.Pawn(color="W"),)\*8,

    tuple(pieces\_mod.Pawn(color="W") for \_ in range(8)),

    (

        pieces\_mod.Rook(color="W"),

        pieces\_mod.Knight(color="W"),

        pieces\_mod.Bishop(color="W"),

        pieces\_mod.Queen(color="W"),

        pieces\_mod.King(color="W"),

        pieces\_mod.Bishop(color="W"),

        pieces\_mod.Knight(color="W"),

        pieces\_mod.Rook(color="W")

    )

)

The problematic lines of code are commented out and the fixes are below. It turned out that the issue was that the line:

(pieces\_mod.Pawn(color="B"),)\*8

Was not making 8 individual pawn objects with there own different last move property.

This meant that all the pawns were the same object passed by reference and so once one piece had moved forward 2, none of them were able to move forward 2.

tuple(pieces\_mod.Pawn(color="W") for \_ in range(8)),

The above replacement line of code fixed this by making 8 distinct pawn objects.

Knights cannot take pieces

Another issue I noticed when black box testing the finished product was that knights were unable to take pieces. I believe that I must have broken the code at some point at certainly could take pieces at the end of prototype 2.

I created an automated unit test where I replicated the chess game that I had played and asserted that the move I was expecting (knight takes a piece) was in the legal move:

# this tests for a bug I encountered where a knight couldn't take a piece,

    # I created this test to identify the bug

    # I then modified the knight pieces class and used this test to show that the bug was fixed

    def test\_troubleshoot\_bug\_legal\_moves(self):

        # replicate the previous situation

        board\_state = Board\_State()

        board\_state = board\_state.make\_move(

            Vector(1, 0), Vector(1, 2)

        )

        board\_state = board\_state.make\_move(

            Vector(1, 7), Vector(1, -2)

        )

        board\_state = board\_state.make\_move(

            Vector(6, 0), Vector(-1, 2)

        )

        board\_state = board\_state.make\_move(

            Vector(6, 7), Vector(-1, -2)

        )

        board\_state = board\_state.make\_move(

            Vector(4, 1), Vector(0, 1)

        )

        board\_state = board\_state.make\_move(

            Vector(3, 6), Vector(0, -2)

        )

        # board\_state.print\_board()

        legal\_moves = board\_state.generate\_legal\_moves()

        # check that the knight can make the move

        self.assertTrue(

            (Vector(2, 2), Vector(1, 2)) in legal\_moves

        )

This test failed which confirmed that I could recreate the logic error. I then fixed the issue by modifying the below code form the Knight Class

    def generate\_movement\_vectors(self, pieces\_matrix, position\_vector):

        # this function yields all 8 possible vectirs

        def possible\_movement\_vectors():

            vectors = (Vector(2, 1), Vector(1, 2))

            # for each x multiplier, y multiplier and vector combination

            for i\_multiplier, j\_multiplier, vector in iter\_product((-1, 1), (-1, 1), vectors):

                # yield corresponding vector

                yield Vector(

                    vector.i \* i\_multiplier,

                    vector.j \* j\_multiplier

                )

        # iterate through movement vectors

        for movement\_vector in possible\_movement\_vectors():

            # get resultant

            resultant\_vector = position\_vector + movement\_vector

            # look at contents of square

            contents = self.examine\_position\_vector(position\_vector=resultant\_vector, pieces\_matrix=pieces\_matrix)

            # if square is empty yield vector

            # old code that caused the logic error

            # if contents == "empty":

            #     yield movement\_vector

            # corrected code to fix knight bug

            if contents in ("empty", "enemy"):

                yield movement\_vector

You can see at the bottom that I was only returning the movement vector if the contents of the to square was empty. The below code fixed this issue by also allowing the 2 square to contain an enemy piece.

Concurrent workers not caching

I also identified an issue with the worker functions in the parallel minimax module not each individually adding there searches to the database. This means that a parallel depth 2 search of the starting positions produced only 1 cache item with a depth of 2 and no additional depth 1 cache.

This would have prevented the chess program being able to learn (one of the items of my design criteria).

I realised that I had forgotten to add the caching to these worker classes. As a result I changed them to add database caching.

# # this object represents a job that a concurrent worker should complete,

# # I used an object as a function cannot be sent between workers

# class Minimax\_Sub\_Job:

#     # contractor: configure bot by setting these parameters as properties

#     def \_\_init\_\_(self, move\_engine, board\_state, depth, args, kwargs, max\_time=None) -> None:

#         self.move\_engine: Move\_Engine = move\_engine

#         self.board\_state: Board\_State = board\_state

#         # # not sure why depth sometimes is a single length tuple containing an int

#         # # here is a quick fix

#         # if isinstance(depth, tuple):

#         #     if len(depth) == 1:

#         #         depth = depth[0]

#         # assert isinstance(depth, int)

#         self.depth = depth

#         self.max\_time = max\_time

#         # these are other erroneous argument that could be passed to the minimax call

#         self.args = args

#         self.kwargs = kwargs

#     def \_\_call\_\_(self, legal\_moves\_sub\_array):

#         # perform part of a minimax search using a sub set of legal moves

#         # print(f"running minimax job with legal moves sub array: (hash={hash(str(legal\_moves\_sub\_array))})")

#         # print(f"STARTING sub job on moves segment:   {hash(str(legal\_moves\_sub\_array))}")

#         # print(legal\_moves\_sub\_array),

#         result = self.move\_engine.minimax(

#             board\_state=self.board\_state,

#             depth=self.depth,

#             legal\_moves\_to\_examine=legal\_moves\_sub\_array,

#             # is\_time\_expired = self.is\_time\_expired,

#             max\_time=self.max\_time,

#             \*self.args,

#             \*\*self.kwargs

#         )

#       # print(f"FINISHING sub job on moves segment:   {hash(str(legal\_moves\_sub\_array))}")

#         # print(f"Minimax sub job finished (hash={hash(str(legal\_moves\_sub\_array))})")

#         return result

# this object represents a job that a concurrent worker should complete,

# I used an object as a function cannot be sent between workers

class Minimax\_Sub\_Job:

    # contractor: configure bot by setting these parameters as properties

    def \_\_init\_\_(self, board\_state, depth, args, kwargs, additional\_depth, cache\_allowed, max\_time=None) -> None:

        # create a cache manager as necessary

        self.cache\_allowed = cache\_allowed

        self.cache\_manager = DB\_Cache() if cache\_allowed else None

        # create a move engine as necessary

        self.move\_engine: Move\_Engine = Move\_Engine(

            cache\_manager=self.cache\_manager,

            cache\_allowed=cache\_allowed,

            additional\_depth=additional\_depth,

        )

        # assert isinstance(self.move\_engine, Move\_Engine), f"Minimax\_Sub\_Job.\_\_init\_\_    self.move\_engine of unexpected type {type(self.move\_engine)}\n{self.move\_engine!r}"

        # assign parameters as properties

        self.board\_state: Board\_State = board\_state

        self.depth = depth

        self.max\_time = max\_time

        # these are other erroneous argument that could be passed to the minimax call

        self.args = args

        self.kwargs = kwargs

    def \_\_call\_\_(self, legal\_moves\_sub\_array):

        # perform part of a minimax search using a sub set of legal moves

        # print(f"running minimax job with legal moves sub array: (hash={hash(str(legal\_moves\_sub\_array))})")

        # print(f"STARTING sub job on moves segment:   {hash(str(legal\_moves\_sub\_array))}")

        # print(legal\_moves\_sub\_array)d\_state,

        # result = self.move\_engine.minimax(

        # assert isinstance(self.move\_engine, Move\_Engine), f"Minimax\_Sub\_Job.\_\_call\_\_    self.move\_engine of unexpected type {type(self.move\_engine)}\n{self.move\_engine!r}"

        if self.cache\_allowed:

            with self.cache\_manager:

                result = self.move\_engine.minimax(

                    board\_state=self.board\_state,

                    depth=self.depth,

                    legal\_moves\_to\_examine=legal\_moves\_sub\_array,

                    # is\_time\_expired = self.is\_time\_expired,

                    max\_time=self.max\_time,

                    \*self.args,

                    \*\*self.kwargs

                )

        else:

            result = self.move\_engine.minimax(

                board\_state=self.board\_state,

                depth=self.depth,

                legal\_moves\_to\_examine=legal\_moves\_sub\_array,

                # is\_time\_expired = self.is\_time\_expired,

                max\_time=self.max\_time,

                \*self.args,

                \*\*self.kwargs

            )

      # print(f"FINISHING sub job on moves segment:   {hash(str(legal\_moves\_sub\_array))}")

        # print(f"Minimax sub job finished (hash={hash(str(legal\_moves\_sub\_array))})")

        return result

# # this bot is responsible for performing a low depth search of a move to be used to presort the moves

# class Presort\_Moves\_Sub\_Job:

#     # construct object

#     # def \_\_init\_\_(self, depth, board\_state: Board\_State, move\_engine: Move\_Engine,max\_time=None) -> None:

#         # cache manager parameter may not be needed if the cache manager object is also bound to the move engine

#         self.depth = depth

#         self.board\_state = board\_state

#         self.move\_engine = move\_engine

#         self.is\_time\_expired = is\_time\_expired

#         self.max\_time = max\_time

#     def \_\_call\_\_(self, move):

#         # self interrupting

#         if is\_time\_expired(self.max\_time):

#             raise TimeOutError()

#         # score a child, use minimax first call to add caching

#         child = self.board\_state.make\_move(\*move)

#         child\_score, \_ = self.move\_engine.minimax(

#             board\_state=child,

#             depth=self.depth,

#             # is\_time\_expired = self.is\_time\_expired,

#             max\_time=self.max\_time,

#         )

#         return (child\_score, move)

# this bot is responsible for performing a low depth search of a move to be used to presort the moves

class Presort\_Moves\_Sub\_Job:

    # construct object

    def \_\_init\_\_(self, depth, board\_state: Board\_State, cache\_allowed, max\_time=None) -> None:

    # def \_\_init\_\_(self, depth, board\_state: Board\_State, move\_engine: Move\_Engine, cache\_allowed, cache\_manager, max\_time=None) -> None:

        # cache manager parameter may not be needed if the cache manager object is also bound to the move engine

        # assign parameters as properties

        self.depth = depth

        self.board\_state = board\_state

        # create a cache manager as necessary

        self.cache\_allowed = cache\_allowed

        self.cache\_manager = DB\_Cache() if cache\_allowed else None

        # create a move engine object

        self.move\_engine = Move\_Engine(

            presort\_moves=True,

            additional\_depth=0,

            cache\_allowed=self.cache\_allowed,

            cache\_manager=self.cache\_manager,

            # cache\_allowed=self.cache\_allowed,

            # cache\_manager=self.cache\_manager,

        )

        # self.cache\_allowed = cache\_allowed

        # self.cache\_manager = cache\_manager

        self.is\_time\_expired = is\_time\_expired

        self.max\_time = max\_time

    def \_\_call\_\_(self, move):

        # cache manager already built into the minimax move engine first call method

        # print(f"Presort job using cache manager:   {self.cache\_manager!r}")

        # with self.cache\_manager:

        #     child = self.board\_state.make\_move(\*move)

        #     child\_score, \_ = self.move\_engine.minimax\_first\_call(

        #         board\_state = child,

        #         depth = self.depth

        #     )

        #     # print("Closing cache")

        #     result = (child\_score, move)

        # return result

        # self interrupting

        if is\_time\_expired(self.max\_time):

            raise TimeOutError()

        # score a child, use minimax first call to add cache manager being opened twice

        # no need to say no parallel as a basic move engine is used

        child = self.board\_state.make\_move(\*move)

        if self.cache\_allowed:

            with self.cache\_manager:

                # child\_score, \_ = self.move\_engine.minimax\_first\_call(

                child\_score, \_ = self.move\_engine.minimax(

                    board\_state=child,

                    depth=self.depth,

                    # is\_time\_expired = self.is\_time\_expired,

                    max\_time=self.max\_time,

                )

        else:

            # child\_score, \_ = self.move\_engine.minimax\_first\_call(

            child\_score, \_ = self.move\_engine.minimax(

                board\_state=child,

                depth=self.depth,

                # is\_time\_expired = self.is\_time\_expired,

                max\_time=self.max\_time,

            )

        return (child\_score, move)

You can see the old Job classes without caching are commented out. The new Job classes that replaced them were able to concurrently access the database in order to cache their results.

Here is a screenshot of the database after the fix to show that the issue was corrected.

Graphical user interface, text

Description automatically generated

The above database content shows the cache produced by 30 second of thinking by the computer (after I made the first move). As you can now see, the depth 2 search result has been saved to the cache as well as the depth 1 search that were completed in the process.

This means that if instead, 6 users on medium difficulty gave the computer 6 lots of 5 seconds to think about a move, it should now still be able to complete a depth 2 search. This is because it could make progress and by completing a group of depth 1 searches, then stop and start where it left of later.

Erroneous Results:

I found an example of an edge case where my minimax function produced an invalid output:

![Graphical user interface

Description automatically generated]()

Here you can see that I am playing the chess AI and have got it into check. The AI failed to respond with a move and failed to declare an that the game was over. This was due to an internal server error form the following invalid minimax result:



The value 1\_000\_001 is used as a starting value for alpha and is an invalid score to return.

I was able to replicate the error by loading the data from the web client into a unit test and repeating the search:

    def test\_specific\_bug\_null\_move(self):

        pieces\_matrix = [

            [Rook("B"), King("B"), None, None, None, None, Queen("W"), None],

            [None, Pawn("B"), Pawn("B"), None, None, None, None, None],

            [Pawn("B"), None, None, None, None, None, None, None],

            [None, None, None, Pawn("W"), None, None, None, None],

            [None, None, None, None, None, Bishop("W"), None, None],

            [None, None, Pawn("W"), None, None, None, None, None],

            [Pawn("W"), None, None, None, None, Pawn("W"), None, Pawn("W")],

            [None, Rook("W"), None, None, None, King("W"), None, None]

        ]

        next\_to\_go = "B"

        board\_state = Board\_State(next\_to\_go, pieces\_matrix)

        print()

        board\_state.print\_board()

        move\_engine = Move\_Engine\_Prime()

        move\_engine.cache\_allowed = False

        move\_engine.cache\_manager = None

        move\_engine.parallel = False

        # self.assertRaises(

        #     AssertionError,

        #     lambda: move\_engine(board\_state, depth=4)

        # )

        def absolute(x): return x if x >= 0 else 0-x

        # \_, move = move\_engine(board\_state, depth=1)

        # self.assertTrue(move is not None)

        # board\_state = board\_state.make\_move(\*move)

        # print()

        # board\_state.print\_board()

        # \_, move = move\_engine(board\_state, depth=1)

        # self.assertTrue(move is not None)

        # board\_state = board\_state.make\_move(\*move)

        # print()

        # board\_state.print\_board()

        # \_, move = move\_engine(board\_state, depth=1)

        # self.assertTrue(move is not None)

        # board\_state = board\_state.make\_move(\*move)

        # print()

        # board\_state.print\_board()

        print(board\_state.is\_game\_over\_for\_next\_to\_go())

        print(list(board\_state.generate\_legal\_moves()))

        score, move = move\_engine(board\_state, depth=4)

        self.assertTrue(

            absolute(score) < 1\_000\_001 and move is not None,

            msg=repr({"score": score, "move": move})

        )

The following test helped me reproduce the error but not diagnose it.

I then checked that the game wasn’t over in another test (the black king should have one move)

 # this test is to ensure that the game over function can correctly identify a game over situation

    # The program crashed in a way that hasn't happened since (I must have been tinkering with the minimax code)

    # One of my assertion statements triggered and identified that the legal move variable was none.

    # this ensured that the program could successfully identify that the game wasn't over and that the king has one move

    # to correct the believed source of the bug, I ensured timed calls always perform at least a depth 1 search to ensure that a move is always produced.

    def test\_troubleshoot\_bug\_game\_over(self):

        pieces\_matrix = deserialize\_client\_pieces\_matrix(

            json.loads(

                """

                    [[[null,null],["B","♚"],[null,null],[null,null],[null,null],[null,null],[null,null],["B","♜"]],[["W","♜"],[null,null],[null,null],[null,null],["W","♛"],["B","♟︎"],["B","♟︎"],["B","♟︎"]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[[null,null],[null,null],[null,null],[null,null],["B","♟︎"],["W","♝"],[null,null],[null,null]],[[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null],[null,null]],[[null,null],["W","♟︎"],[null,null],[null,null],["W","♚"],[null,null],["W","♟︎"],["W","♟︎"]],[[null,null],[null,null],[null,null],[null,null],[null,null],["W","♝"],["W","♜"],[null,null]]]

                """

            )

        )

        board\_state = Board\_State(

            next\_to\_go="B",

            pieces\_matrix=pieces\_matrix,

            check\_encountered=True

        )

        # board\_state.print\_board()

        legal\_moves = list(board\_state.generate\_legal\_moves())

        # print({"legal\_moves": legal\_moves})

        self.assertEqual(

            legal\_moves,

            [(Vector(i=1, j=7), Vector(i=1, j=0))]

        )

        over, \_ = board\_state.is\_game\_over\_for\_next\_to\_go()

        # print(over)

        self.assertTrue(not over)

This test work and passed without issue. This meant that the issue was with the minimax function as my chess engine could successfully identify that there was one remaining legal move.

I didn’t figure out what caused the issue but I added checks within my code as part of a defensive design strategy to prevent the issue causing unexpected behaviour.

I added the following code to the Timed\_Move\_Engine:

        # while there is time left, keep searching to a greater depth

        while not time\_used\_up():

            # print(f"Iterating again: time\_used\_up()   -->   {time\_used\_up()}")

            try:

                # deepest\_result = self.minimax\_first\_call\_parallel(board\_state=board\_state, depth=depth, is\_time\_expired=time\_used\_up)

                # search the tree in parallel if needed

                result = self.minimax\_first\_call\_parallel(board\_state=board\_state, depth=depth, max\_time=self.max\_time, variable\_depth=0)

                score, move = result

                # if the move or score are invalid then raise the appropriate error

                if absolute(score) == 1\_000\_001 or move is None:

                    raise UnexplainedErroneousMinimaxResultError()

            except UnexplainedErroneousMinimaxResultError:

                # I don't know the cause of the error but I can catch it here and prevent it causing issues and producing unexpected behaviour

                break

            except TimeOutError:

                # print(f"Breaking: time\_used\_up()   -->   {time\_used\_up()}")

                break

            else:

                # print(f"Completes depth={depth} so incrementing depth")

                # print(f"Result at depth {depth}:   {deepest\_result}")

                # if neither the time is used up or an erroneous result was produced, keep searching at a greater depth

                deepest\_result = result

                depth += 1

The idea was that I would detect this invalid result and not use it, using the previous result.

I also added code to the cache manager to prevent invalid results from being added to the database cache:

    def add\_to\_cache(self, board\_state: Board\_State, score, depth, move):

        # this function adds to the cache

        # this function assumes that no move valuable cache already exists (it overwrites all other cache)

        assert self.engaged, "Context manager must be used"

        # this is used to tackle any bugs elsewhere in the program

        # it ensures that no invalid cache items are added to the database

        def absolute(x): return x if x >= 0 else 0-x

        if absolute(score) > 1\_000\_000 or (move is None and depth > 0):

            # don't add erroneous data to cache

            return None

These 2 combined changes fixed the issue and prevented the unexpected behaviour from causing the program to crash or from producing an invalid output.