**Prototype 3 log**

Again I am keeping of log of how this prototype is going. This log will from now on include more screenshots to help me to write the final documentation.

The first thing I will do if refactor the minimax function. I will break it up into multiple parts to reduce it complexity. I will then perform speed tests to assert that some of the changes I have implemented have improved performance.

I will try to extract out some of the arguments that are more to do with setup and create an object and constructor or a factory object. I could also add caching in memory if I used an object. I will use a jupyter notebook and the time library to ensure it is working.

Here was my initial idea

class Move\_Engine():

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

        assert self.memoization\_cache is not None

        self.memoization\_cache[(board\_state, depth)] = (score, move)

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

        assert self.memoization\_cache is not None

        return self.memoization\_cache.get((board\_state, depth))

    def \_\_init\_\_(self, variable\_depth: bool = True, presort\_moves: bool = True, color\_maximizer\_key: dict | None = None, memoization: bool = True) -> None:

        self.variable\_depth = variable\_depth

        self.presort\_moves = presort\_moves

        if color\_maximizer\_key is not None:

            self.color\_maximizer\_key = color\_maximizer\_key

        else:

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

        if memoization:

            self.memoization\_cache = dict()

        else:

            self.memoization\_cache = None

    def \_\_call\_\_(self, board\_state, depth) -> tuple[int, tuple[Vector, Vector]]:

        # returns score, move

        pass

I need to implement the cache differently as I want to check if a search has been made on the given board state to a depth of the given depth or greater.

My cache will be implemented as a hash table. The key will be the hash of the board state and the result will be the greatest depth search that has occurred.

This is what I have now created

class Move\_Engine():

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

        assert self.memoization\_cache is not None

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

        needs\_update: bool

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

            needs\_update = True

        else:

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

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

        if needs\_update:

            self.memoization\_cache[board\_state] = {

                depth: depth,

                score: score,

                move: move

            }

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

        assert self.memoization\_cache is not None

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

            return None

        else:

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

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

            if record\_useful:

                return self.memoization\_cache[board\_state]

            else:

                return None

    def \_\_init\_\_(self, variable\_depth: bool = True, presort\_moves: bool = True, color\_maximizer\_key: dict | None = None, memoization: bool = True) -> None:

        self.variable\_depth = variable\_depth

        self.presort\_moves = presort\_moves

        if color\_maximizer\_key is not None:

            self.color\_maximizer\_key = color\_maximizer\_key

        else:

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

        if memoization:

            self.memoization\_cache = dict()

        else:

            self.memoization\_cache = None

    def minimax(self, board\_state, depth, alpha=ARBITRARILY\_LARGE\_VALUE, beta=0-ARBITRARILY\_LARGE\_VALUE) -> tuple[int, tuple[Vector, Vector]]:

        # returns score, move

        cached\_result = self.search\_cache(board\_state=board\_state, depth=depth)

        if cached\_result is not None:

            return cached\_result["score"], cached\_result["move"]

        else:

            # put code for minimax function here

            # use self.color\_maximizer\_key and board\_state.next\_to\_go to get is maximizer

            score, move = None, None

            self.add\_to\_cache(

                board\_state=board\_state,

                depth=depth,

                score=score,

                move=move

            )

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

        return self.minimax(\*args, \*\*kwargs)

I still have to decide if I want to cache calls at all depths or not bother with low depths like 1 or 0.

It won’t be a lot of space to store them though: the hash, depth, score, movement vector are all really small.

I have written a function to return child game states:

    def generate\_child\_game\_states\_and\_moves(self, board\_state: Board\_State, depth):

        if not self.presort\_moves or depth <= 1:

            for move in board\_state.generate\_legal\_moves():

                child\_board\_state: Board\_State = board\_state.make\_move(move)

                yield move, child\_board\_state

        else:

            def get\_move\_score\_and\_child(move):

                child\_board\_state = board\_state.make\_move(move)

                score = self.minimax(

                    board\_state=child\_board\_state,

                    variable\_depth=False

                )

                return move, child\_board\_state, score

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

            return sorted(

                map(

                    get\_move\_score\_and\_child,

                    board\_state.generate\_legal\_moves()

                ),

                key = lambda items: items[2],

                # descending order for maximizer

                reverse = is\_maximizer

            )

It is affected by the pre-sort attribute and it uses a version of minimax with caching.

I then wrote a base case function

    def base\_case(self, board\_state: Board\_State, alpha, beta):

        # returns score: int, move: (V, V) | None

        # used then depth == 0 or game over

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

        in\_check = board\_state.color\_in\_check()

        if self.variable\_depth and not is\_over and in\_check:

            return self.minimax(

                board\_state=board\_state,

                depth = self.ADDITIONAL\_DEPTH,

                alpha=alpha,

                beta=beta

            )

        else:

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

I then used these components to create my minimax function. This left the class as the following:

# 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

class Move\_Engine():

    ADDITIONAL\_DEPTH = 2

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

        assert self.memoization\_cache is not None

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

        needs\_update: bool

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

            needs\_update = True

        else:

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

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

        if needs\_update:

            self.memoization\_cache[board\_state] = {

                depth: depth,

                score: score,

                move: move

            }

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

        assert self.memoization\_cache is not None

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

            return None

        else:

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

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

            if record\_useful:

                return self.memoization\_cache[board\_state]

            else:

                return None

    def \_\_init\_\_(self, variable\_depth: bool = True, presort\_moves: bool = True, color\_maximizer\_key: dict | None = None, memoization: bool = True) -> None:

        self.variable\_depth = variable\_depth

        self.presort\_moves = presort\_moves

        if color\_maximizer\_key is not None:

            self.color\_maximizer\_key = color\_maximizer\_key

        else:

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

        if memoization:

            self.memoization\_cache = dict()

        else:

            self.memoization\_cache = None

    def generate\_move\_child(self, board\_state: Board\_State, depth):

        # returns move, child

        if not self.presort\_moves or depth <= 1:

            for move in board\_state.generate\_legal\_moves():

                child\_board\_state: Board\_State = board\_state.make\_move(move)

                yield move, child\_board\_state

        else:

            def get\_move\_score\_and\_child(move):

                child\_board\_state = board\_state.make\_move(move)

                score = self.minimax(

                    board\_state=child\_board\_state,

                    variable\_depth=False

                )

                return move, child\_board\_state

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

            return sorted(

                map(

                    get\_move\_score\_and\_child,

                    board\_state.generate\_legal\_moves()

                ),

                key = lambda items: items[2],

                # descending order for maximizer

                reverse = is\_maximizer

            )

    def base\_case(self, board\_state: Board\_State, alpha, beta):

        # returns score: int, move: (V, V) | None

        # used then depth == 0 or game over

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

        in\_check = board\_state.color\_in\_check()

        # base case has a secret extra recursive case

        if self.variable\_depth and not is\_over and in\_check:

            return self.minimax(

                board\_state=board\_state,

                depth = self.ADDITIONAL\_DEPTH,

                alpha=alpha,

                beta=beta

            )

        else:

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

    def minimax(self, board\_state: Board\_State, depth, alpha=ARBITRARILY\_LARGE\_VALUE+1, beta=-(ARBITRARILY\_LARGE\_VALUE+1), variable\_depth: bool | None = None) -> tuple[int, tuple[Vector, Vector]]:

        # returns score, move

        if variable\_depth is None:

            variable\_depth = self.variable\_depth

        cached\_result = self.search\_cache(board\_state=board\_state, depth=depth)

        if cached\_result is not None:

            return cached\_result["score"], cached\_result["move"]

        # put code for minimax function here

        # use self.color\_maximizer\_key and board\_state.next\_to\_go to get is maximizer

        score, move = None, None

        # call base case:

        if depth == 0 or board\_state.is\_game\_over\_for\_next\_to\_go():

            return self.base\_case(

                board\_state=board\_state,

                alpha=alpha,

                beta=beta

            )

        best\_move: tuple[Vector] | None = None

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

        best\_score: int | None = None

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

        if is\_maximizer:

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

            for move, child\_board\_state in self.generate\_move\_child(board\_state=board\_state, depth=depth):

                score = self.minimax(

                    board\_state=child\_board\_state,

                    depth=depth-1,

                    alpha=alpha,

                    beta=beta

                )

                if score > best\_score:

                    alpha = max(alpha, score)

                    best\_score = score

                    best\_move = move

                    best\_child\_board\_state = child\_board\_state

                if beta <= alpha:

                    # prune

                    break

        else:

            best\_score = ARBITRARILY\_LARGE\_VALUE+1

            for move, child\_board\_state in self.generate\_move\_child(board\_state=board\_state, depth=depth):

                score = self.minimax(

                    board\_state=child\_board\_state,

                    depth=depth-1,

                    alpha=alpha,

                    beta=beta

                )

                if score < best\_score:

                    beta = min(beta, score)

                    best\_score = score

                    best\_move = move

                    best\_child\_board\_state = child\_board\_state

                if beta <= alpha:

                    # prune

                    break

        self.add\_to\_cache(

            board\_state=best\_child\_board\_state,

            depth=depth,

            score=best\_score,

            move=best\_move

        )

        return score, move

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

        return self.minimax(\*args, \*\*kwargs)

The next thing that I want to do is to change where the 3 repeat game states logic is occurring. I want it to occur at a lower level in the board state class so that the minimax algorithm is aware of it.

I also want to separate the logic and the abstract interface from the pieces class. I also need to fix the move foreword 2 move for the pawn.

I also want to disentangle the cache from the minimax by again using inheritance. This will allow me to interchange caching from physical memory and a database.

I have worked on my code and have achieved the following things:

* Moved 3 repeat stalemate logic from game state to board state class. This still requires further testing.
* Fixed the pawn moving forward 2 issue. Still needs a corresponding unit tests to be added.
* Attempted to debug some of the issues with my minimax function

I believe that my minimax function is broadly functioning but the cache is not working. Let me try to fix this.

I have attempted to create a new minimax test to verify that my performance increases are actually working. The results are interesting.

I created a test where I compared 2 different move engine objects, one with pre-sorting and one without. I controlled for caching and variable depth by turning them off. I added a counter to the move engine to count how many times alpha beta pruning occurred.

I created a 2 tests and ran them for depths of 2 and 3. They tested that the pre-sort was faster (using time.perfcounter ) and that it was making more prunes. I also asserted that the output was the same. The results were as follows.

Text

Description automatically generated

The pre-sort seemed to be slower and have less prunes. I was curious to see the effect of reversing the pre-sort order. This caused the test to fail in a different way as the functions returned a different value?

Text

Description automatically generated

This requires further investigation in a notebook.

This issue was identified. The difference was that the functions were returning slightly different moves with the same score (move rook).

I then added to the test code and refactored it to make if readable. It now allows me to test a variety of mutations of the move engine by perf count or static evals.

I found that the pre-sort optimisation did improve if it was set to a depth of max(0, depth-3). I also made a test for the json cache. I have the cache tested against without cache on a mutant of the starting position with some number of random moves.

Due to the read write times of the json file, the performance is only improved for some depths.

I have had some issues with the caching tests. I created a jupyter notebook to perform searches of the starting board state at depths of 1, 2, 3 and 4 with caching. I then check that the tests run faster if I restart the notebook and run a second time due to the persistent json cache. I then had an issue where my unit tests were performing differently each time I ran them:

Text

Description automatically generated

Unit tests:

With trials = 1 (one core used) the results are as expected:

Text

Description automatically generated

With multiple trials is all goes a bit weird:

Trials = 8

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

As we can see something is going wrong. The cache seems to every now and then be lost meaning the result must be found manually and the cache overwritten (hence cache goes from depth 4 to 2)

Perhaps I could have all of the workers separate move engines share the same cache handler so that there are no instances where the file isn’t read correctly as it is being opened by another thread.

Workers = 4 + many print statements + singleton json cache object used

Text

Description automatically generated

This shows the import of move engine is happening an additional time for each thread. This caused the json cache to still be initialised multiple times causing issues. The solution would be to try a different way to ensure that at no point in the parallel threads are any files read.

Let me see…

Ok so I have developed what I thought was a solution but instead it just exemplifies the issue (still progress).

For the test time minimax I devised a test to verify that the cache was improving performance:

class Job\_test\_cache\_vs\_normal:

    def \_\_init\_\_(self, moves, check\_time, cache\_manager, depth) -> None:

        self.moves = moves

        self.check\_time = check\_time

        self.cache\_manager = cache\_manager

        self.depth = depth

    def \_\_call\_\_(self, \_):

        board\_state = Random\_Mutant\_Generator()(moves=self.moves)

        try:

            Check\_Benchmarks()(

                game\_engine\_arguments={

                    "board\_state": board\_state,

                    "depth": self.depth

                },

                good\_move\_engine=Move\_Engine(variable\_depth=False, use\_validator=True, presort\_moves=True, cache\_allowed=True, cache\_manager=self.cache\_manager),

                bad\_move\_engine=Move\_Engine(variable\_depth=False, use\_validator=True, presort\_moves=True, cache\_allowed=False, cache\_manager=self.cache\_manager),

                check\_time=self.check\_time,

                check\_evals=True,

                check\_move=False

            )

        except AssertionError as e:

            # self.test\_case.fail(msg=str(e))

            return False, str(e)

        else:

            return True, None

    def test\_cache\_vs\_normal\_depth\_3(self):

        trials = 16

        pass\_bar = 12

        argument\_iterable = (None,)\*trials

        cache\_manager = JSON\_cache\_manual\_save()

        job = Job\_test\_cache\_vs\_normal(

            moves=10,

            check\_time=True,

            cache\_manager=cache\_manager,

            depth=3

        )

        results = Multitasker()(

            job=job,

            argument\_iterable=argument\_iterable

        )

        # assume multitasker is return not yield so all threads now finished

        cache\_manager.save()

        successes = sum(int(success) for success, \_ in results)

        if successes < pass\_bar:

            error\_msgs = "\n\t".join(msg for success, msg in results if not success)

            # error\_msgs = "".join(msg for success, msg in results if not success)

            self.fail(f"\n{successes}/{trials} were more efficient with cache (bar was {pass\_bar}):\n{error\_msgs}")

        # for success, msg in results:

        #     self.assertTrue(

        #         success,

        #         msg=msg

        #     )

    def test\_cache\_vs\_normal\_depth\_2\_less\_static\_evals(self):

        trials = 16

        pass\_bar = 12

        argument\_iterable = (None,)\*trials

        cache\_manager = JSON\_cache\_manual\_save()

        job = Job\_test\_cache\_vs\_normal(

            moves = 10,

            check\_time = False,

            cache\_manager = cache\_manager,

            depth = 2

        )

        results = Multitasker()(

            job=job,

            argument\_iterable = argument\_iterable

        )

        # assume multitasker is return not yield so all threads now finished

        cache\_manager.save()

        successes = sum(int(success) for success, \_ in results)

        if successes < pass\_bar:

            error\_msgs = "\n\t".join(msg for success, msg in results if not success)

            # error\_msgs = "".join(msg for success, msg in results if not success)

            self.fail(f"\n{successes}/{trials} were more efficient with cache (bar was {pass\_bar}):\n{error\_msgs}")

        # for success, msg in results:

        #     self.assertTrue(

        #         success,

        #         msg=msg

        #     )

This all revolved around a fix to the cache:

I created a cache manager that didn’t autosave

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

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

        pass

I also made the actual cache manager object a parameter for the move engine function.

    def \_\_init\_\_(self, cache\_manager: RAM\_cache| None = JSON\_Cache(), cache\_allowed: bool = True, variable\_depth: bool = True, additional\_depth=1, presort\_moves: bool = True, color\_maximizer\_key: dict | None = None, use\_validator: bool = False) -> None:

I then created some code to try to build up my cache:

import multiprocessing

from random import randint, choice as random\_choice

import time

from minimax import Move\_Engine, JSON\_Cache\_Manual\_Save

from board\_state import Board\_State

def random\_board\_state(moves: int, seed\_board\_state=None):

    if seed\_board\_state is None:

        seed\_board\_state = Board\_State()

    board\_state = seed\_board\_state

    for \_ in range(moves):

        random\_move = random\_choice(list(board\_state.generate\_legal\_moves()))

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

    return board\_state

def multi\_processor(job, argument\_iterable):

    cores = multiprocessing.cpu\_count()

    with multiprocessing.Pool(cores) as pool:

        return pool.map(

            func=job,

            iterable=argument\_iterable

        )

print("creating json cache manager manual save")

cache\_manager = JSON\_Cache\_Manual\_Save()

move\_engine = Move\_Engine(

    cache\_allowed=True,

    cache\_manager=cache\_manager,

    variable\_depth=False

)

def job(job\_data):

    depth, moves = job\_data["depth"], job\_data["moves"]

    start = time.perf\_counter()

    move\_engine(

        board\_state=random\_board\_state(moves),

        depth=depth

    )

    return time.perf\_counter() - start

def generate\_argument\_iterable(batch\_size, depth, moves\_range):

    for \_ in range(batch\_size):

        yield {

            "depth": depth,

            "moves": random\_choice(moves\_range)

        }

def execute\_batch\_and\_save(batch\_size, depth, moves\_range):

    argument\_iterable = generate\_argument\_iterable(batch\_size=batch\_size, depth=depth, moves\_range=moves\_range)

    previous\_cache\_size = move\_engine.cache\_manager.cache\_size()

    print(f"STARTING:   execute\_batch\_and\_save(batch\_size={batch\_size}, depth={depth}, moves\_range={moves\_range})")

    times = multi\_processor(

        job=job,

        argument\_iterable=argument\_iterable

    )

    mean\_time = sum(times) / len(times)

    print(f"FINISHED:   execute\_batch\_and\_save(batch\_size={batch\_size}, depth={depth}, moves\_range={moves\_range})")

    print(f"Mean time:   {mean\_time}")

    new\_cache\_size = move\_engine.cache\_manager.cache\_size()

    new\_cache = move\_engine.cache\_manager.memoization\_cache

    print(f"New cache:\n{new\_cache}")

    print(f"Saving {new\_cache\_size - previous\_cache\_size} new cache items...")

    cache\_manager.save()

    print("Cache saved to json")

def main():

    execute\_batch\_and\_save(batch\_size=2, depth=2, moves\_range=range(1))

    # execute\_batch\_and\_save(batch\_size=1, depth=4, moves\_range=range(1))

    # execute\_batch\_and\_save(batch\_size=40, depth=2, moves\_range=range(1, 11))

    # execute\_batch\_and\_save(batch\_size=16, depth=3, moves\_range=range(1, 11))

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

    main()

# def job(i): return f"HI:  {i}"

# cores = multiprocessing.cpu\_count()

# def main():

#     with multiprocessing.Pool(cores) as p:

#         result = (

#             p.map(

#                 job,

#                 range(8)

#             )

#         )

#     print(result)

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

#     main()

This resulted in an issue. Here I will highlight is with one trial

Text

Description automatically generated

Then 2 trials:

A picture containing application

Description automatically generated

The code is structured such that it appears that a json cache manual is initialised, shared between workers and then saved once at the end. However despite this the json cache in being created many times. Each worker has its own json cache and the original one that is manually saved contains none of the cache data.

From this I now know that I cannot rely on workers sharing memory / objects. No matter how it appears they will use a separate copy. This makes my cache method incompatible with a persistent JSON storage strategy as a json file cannot be accessed simultaneously by many threads in parallel. I will now investigate if a SQL database could be used and shared between many threads. I would need to keep the ORM setup minimal to minimise overheads.

It has been a while since I created an update. Here is that I have done since.

I did indeed create a database system to use as a cache. This used marshmallow schemas and SQL-Alchemy as the ORM. I found that using the separate libraries was easier. I used a hybrid approach where depth 0 cache is stored in ram only during a session with the cache manages (within a context manager). The depth 1 + cache is stored in a persistent local database file.

I also perfected the parallel minimax function. It works be dividing a depth n job with all its legal moves into many depth n jobs with a portion of these legal moves. These jobs are then run in parallel by different workers.

I also pursued a variety of other optimisations by running games in the console. I hope soon to make a test to prove the results.

I tried to add LRU caching to all the functions I could. I realised that sometimes with caused unit tests to fail and so I used the test to decide where to add the cache. Unfortunately this seemed to slow the program down and so the idea was abandoned.

I experimented with determining how many workers is optimal for the parallel minimax function. My logic was that was that is was possible that the answer was less than 8 as less workers could still use all of the CPU but could benefit more form pruning. It seems 8 and 5 work best

I created a function to try to decide if a minimax call should use parallelisation to try to speed up the function by having it be most efficient.

I also experimented with using different strategies to divide up the legal moves for the workers in the parallel minimax function. I knew that it was best for the pruning if the legal moves were sorted but I wasn’t sure if I should put the best 4 in one array then the next best 4 in another array and so on or give each array in sorted order a good 2 mediocre and a bad one.

I also experimented with deciding the best depth for pre-sorting and was unsure if I should use max(0, depth-2) or max(0, depth-3).

I ran into an issue where the parallel minimax was sometimes returning a different value to the normal one. I created a test to prove the issue was occurring. I then used a pickle file to store any board states which had this discrepancy. It turned out that the issue was that the worker minimax calls didn’t use the appropriate additional depth setting.

I am currently debugging and issue with my depth 3 vs 2 test where the game seems to result in stalemate. I am using a pickle file to store all the game objects after each move for analysis later with a notebook.

A picture containing text, computer

Description automatically generated

This is what my depth 1 vs 2 test looks like (additional depth 1)

­­­­I have fixed this issue. The issue was that the parallel worker that the parallel minimax algorithm was using was not carrying forward the correct minimum depth settings. This meant that when I was testing parallel vs non-parallel the depth that was being tested was different leading to different outcomes. I have created a test to verify that the same deterministic outcomes are reached be the different minimax algorithms.

I now need to plan out how I will get my user interface to work.

Here is my idea. I to create a webserver with flask. I will use a session object to keep track of the individual user’s game.

I will have the index page be a simple how screen with a start game button. This will forward to a config page where difficulty settings will be selected.

To tackle the inconsistency of my minimax function in terms of time taken I have a plan.

A picture containing text, computer

Description automatically generated­Here is some data to exemplify the issue I have had.

­­

Here we can see that the black player (depth 2) has the issue of