POLITECNICO DI TORINO

01URRSM

Computational Intelligence Final Report

Sidharrth Nagappan

s307031

Contents

1 Introduction

Though I'm an Erasmus student, I had a great time taking this course and have learnt a lot about problem solving algorithms, game theory and reinforcement learning. Above all, I not only learnt from professors, but also from peers that are a lot older than me, and peer reviews really helped. I am grateful to have received a total of 9 detailed peer reviews over the semester.

In labs, I have tried to exceed the set requirements, often experimenting with strategies I read in papers/found online and explaining them thoroughly in my lab READMEs.

This report details my activities throughout the semester, and is a testament to my time in Turin.

2 Lab 1

2.1 Solution

Lab 1 concerned the combinatorial optimisation of the set cover problem, which is NP-hard. The problem is to find a minimum set of subsets of a given set of subsets such that all elements of the given set are covered. Since a solution cannot be found in polynomail time, any implemented solution is guaranteed to be suboptimal. For this lab, the problem is tackled through a collection of search algorithms:

1. Naive Greedy

2. Greedy with a better cost function

3. A* Traversal Using a Priority Queue

4. A* Traversal Using a Fully Connected Graph

2.1.1 Naive Greedy

```python
def naive_greedy(N):
  goal = set(range(N))
  covered = set()
  solution = list()
  all_lists = sorted(problem(N, seed=42), key=lambda l: len(l))
  while goal != covered:
      x = all_lists.pop(0)
      if not set(x) < covered:
          solution.append(x)
          covered |= set(x)

  print(
      f"Naive greedy solution for N={N}: w={sum(len(_) for _ in solution)}
      ↪  (bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
  )
```

The greedy algorithm essentially traverses through a sorted list of subsets and keeps adding the subset to the solution set if it covers any new elements. The algorithm is very naive as it does not take into account the number of new elements.

2.1.2 Greedy with basic heuristic approximation

This version of the greedy algorithm takes the subset with the lowest heuristic f where S_e is the expected solution (containing all the unique elements) and n_i is

the current subset:

$$f_i = 1/|n_i - S_e|$$

In real-life scenarios, the cost depends on the relative price of visiting a node/-choosing an option. Since we consider all options to be arbitrarily priced, we use a constant cost of 1.

```python
def set_covering_problem_greedy(N, subsets, costs):
    cost = 0
    visited_nodes = 0
    already_discovered = set()
    final_solution = []
    expected_solution = set(list(itertools.chain(*subsets)))
    covered = set()
    while covered != expected_solution:
        subset = min(subsets, key=lambda s: costs[subsets.index(s)] /
        ↪ (len(set(s)-covered) + 1))
        final_solution.append(subset)
        cost += costs[subsets.index(subset)]
        visited_nodes = visited_nodes+1
        covered |= set(subset)
    print("NUMBER OF VISITED NODES: ", visited_nodes)
    print("w: ", sum(len(_) for _ in final_solution))
    print(
        f"Naive greedy solution for N={N}: w={sum(len(_) for _ in final_solution)}
        ↪ (bloat={(sum(len(_) for _ in final_solution)-N)/N*100:.0f}%)"
    )
    print(
        f"My solution for N={N}: w={sum(len(_) for _ in final_solution)}
        ↪ (bloat={(sum(len(_) for _ in final_solution)-N)/N*100:.0f}%)"
    )
    return final_solution, cost

for n in [5, 10, 50, 100, 500, 1000]:
    subsets = problem(n, seed=SEED)
    set_covering_problem_greedy(n, subsets, [1]*len(subsets))
```

2.1.3 A* Search Using a Priority Queue

The A* algorithm requires a monotonic heuristic function that symbolises the remaining distance between the current state and the goal state. In the case of the set cover problem, the heuristic function is the number of elements that are not covered by the current solution set, such that finding all unique elements symbolises reaching the goal state. The algorithm is implemented using a priority queue.

The implemented algorithm can be surmised as pseudocode below:

1. Add the start node to the priority queue

2. While the state is not None, cycle through the subsets and compute the cost of adding this subset to the final list.

3. If the cost has not been stored yet and the the new state is not in the queue, update the parent of each state. If travelling in this route produces a cheaper cost, update the cost of the node and its parent.

4. Finally, compute the path we travelled through.

```python
from typing import Callable
from helpers import State, PriorityQueue
import numpy as np

class AStarSearch:
    def __init__(self, N, seed=42):
        # N is the number of elements to expect
        self.N = N
        self.seed = seed

    def add_to_state(self, st, subset):
        '''
        Unnecessary function to add a subset to a state because we are using
        the State class instead of a normal np.array
        '''
        state_list = st.copy_data().tolist()
        state_list.append(subset)
        return State(np.asarray(state_list, dtype=object))

    def are_we_done(self, state):
        '''
        Check if we have reached the goal state (such that all elements are
        covered in range(N))
        '''
        flattened_list = self.flatten_list(state.copy_data().tolist())
        for i in range(self.N):
            if i not in flattened_list:
                return False
        # print("We are done")
        return True

    def flatten_list(self, l):
        '''
        Utility function to flatten a list of lists using itertools
        '''
        return list(itertools.chain.from_iterable(l))

    def h(self, state):
```

```python
            '''
            Heuristic Function h(n) = number of undiscovered elements
            '''
            num_undiscovered_elements = len(set(range(self.N)) -
            ↪   set(self.flatten_list(state.copy_data().tolist())))
            return num_undiscovered_elements

    def astar_search(
            self,
            initial_state: State,
            subsets: list,
            parents: dict,
            cost_of_each_state: dict,
            priority_function: Callable,
            unit_cost: Callable,
        ):
            frontier = PriorityQueue()
            parents.clear()
            cost_of_each_state.clear()

            visited_nodes = 1
            state = initial_state
            parents[state] = None
            cost_of_each_state[state] = 0
            # to find length at the end without needed to flatten the state
            discovered_elements = []

            while state is not None and not self.are_we_done(state):
                for subset in subsets:
                    # if this list has already been collected, skip
                    if subset in state.copy_data():
                        # print("Already in")
                        continue
                    new_state = self.add_to_state(state, subset)
                    state_cost = unit_cost(subset)
                    # if new_state not in cost_of_each_state or
                    ↪   cost_of_each_state[new_state] > cost_of_each_state[state] +
                    ↪   state_cost:
                    if new_state not in cost_of_each_state and new_state not in
                    ↪   frontier:
                        parents[new_state] = state
                        cost_of_each_state[new_state] = cost_of_each_state[state] +
                        ↪   state_cost
                        frontier.push(new_state, p=priority_function(new_state))
                    elif new_state in frontier and cost_of_each_state[new_state] >
                    ↪   cost_of_each_state[state] + state_cost:
                        parents[new_state] = state
                        cost_of_each_state[new_state] = cost_of_each_state[state] +
                        ↪   state_cost
                if frontier:
```

```
80                  state = frontier.pop()
81                  visited_nodes += 1
82              else:
83                  state = None
84
85          path = list()
86          s = state
87
88          while s:
89              path.append(s.copy_data())
90              s = parents[s]
91
92          print(f"Length of final list: {len(self.flatten_list(path[0]))}")
93          print(f"Found a solution in {len(path):,} steps; visited
            ↪ {len(cost_of_each_state):,} states")
94          print(f"Visited {visited_nodes} nodes")
95          print(
96              f"My solution for N={self.N}: w={sum(len(_) for _ in path[0])}
                ↪ (bloat={(sum(len(_) for _ in
                ↪ path[0])-self.N)/self.N*100:.0f}%)"
97          )
98          return list(reversed(path))
99
100     def search(self, constant_cost=False):
101         GOAL = State(np.array(range(self.N)))
102         subsets = problem(self.N, seed=self.seed)
103         initial_state = State(np.array([subsets[0]]))
104
105         parents = dict()
106         cost_of_each_state = dict()
107
108         self.astar_search(
109             initial_state = initial_state,
110             subsets = subsets,
111             parents = parents,
112             cost_of_each_state = cost_of_each_state,
113             priority_function = lambda state: cost_of_each_state[state] +
                ↪ self.h(state),
114             unit_cost = lambda subset: 1 if constant_cost else len(subset)
115         )
```

The unit cost during search can either be set to a constant of 1 or the length of chosen subsets. The latter is employed as it helps the algorithm focus on finding all the elements with minimal overhead (redundant elements).

2.1.4 A* Search with Fully Connected Graph (Failed Idea)

An initial idea I had was to build a fully connected graph where each subset is in it's own node, and run an A* star search to traverse it and find a shortest path.

For several logical and overhead reasons, this idea produced poor results and large bloats for big Ns.

Given A $= [2, 4, 5]$, B $= [2, 3, 1]$ and C $= [1, 2]$,
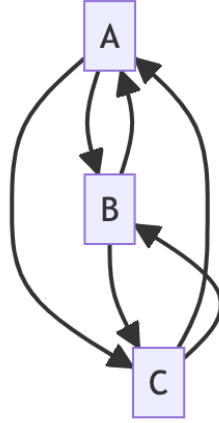
Figure 1: Fully connected graph

The heuristic function is slightly different:

$$h_i = len(s_i) - len(s_i \cap S_e)$$

where s_i is the current subset and S_e is the expected solution. It takes into account both the length of the new subset (to minimise final weight) and the number of undiscovered elements that it can contribute.

We can also immediately return a very large heuristic value such as 100 in the case of duplicating elements in the subset or in any situation where we want a certain node to be immediately skipped.

```python
class AStarSearchFullyConnectedGraph:
    def __init__(self, adjacency_list, list_values, N):
        self.adjacency_list = adjacency_list
        self.list_values = list_values
        H = {}
        for key in list_values:
            # heuristic value is length of list
            H[key] = len(list_values[key])
        self.H = H
        # holds the lists of each visited node
        self.final_list = []
        # N is the count of elements that should be in the final list
        self.N = N
        self.discovered_elements = set()
```

```python
16    def flatten_list(self, _list):
17        return list(itertools.chain.from_iterable(_list))
18
19    def get_neighbors(self, v):
20        return self.adjacency_list[v]
21
22    def get_number_of_elements_not_in_second_list(self, list1, list2):
23        count = 0
24        # flattened_list = self.flatten_list(list2)
25        for i in set(list1):
26            # print("i: ", i)
27            if i not in list2:
28                count += 1
29        # if count > 1:
30        #     print("count: ", count)
31        return len(set(list1) - set(list2))
32
33    # f(n) = h(n) + g(n)
34
35    def h(self, n):
36        num_new_elements =
        ↪  self.get_number_of_elements_not_in_second_list(self.list_values[n],
        ↪  self.discovered_elements)
37        # if self.list_values[n] in self.final_list:
38        #     return 1000
39        return num_new_elements
40        # return self.H[n] / (num_new_elements + 1)
41
42    def get_node_with_least_h(self):
43        min_h = float("inf")
44        min_node = None
45        for node in self.adjacency_list:
46            if self.h(node) < min_h:
47                min_h = self.h(node)
48                min_node = node
49        return min_node
50
51    def get_node_with_least_h_and_not_in_final_list(self):
52        min_h = float("inf")
53        min_node = None
54        for node in self.adjacency_list:
55            if self.h(node) < min_h and node not in self.final_list:
56                min_h = self.h(node)
57                min_node = node
58        return min_node
59
60    # visited_node = [1, 2, 3]
61    # final_list = [[4, 5], [1]]
62    def are_we_done(self):
63        # flattened_list = list(itertools.chain.from_iterable(self.final_list))
```

```python
        for i in range(self.N):
            if i not in self.discovered_elements:
                return False
        print("We are done")
        return True

    def insert_unique_element_into_list(self, _list, element):
        if element not in _list:
            _list.append(element)
        return _list

    def a_star_algorithm(self):
        # start_node is node with lowest cost
        start_node = self.get_node_with_least_h()

        open_list = [start_node]
        closed_list = []

        g = {}

        g[start_node] = 0

        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the highest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) > g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            print(f"Visiting node: {n}")
            self.final_list.append(self.list_values[n])
            # self.discovered_elements.union(self.list_values[n])
            # add list_values[n] to discovered_elements
            for i in self.list_values[n]:
                self.discovered_elements.add(i)
            print(len(self.discovered_elements))

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if self.are_we_done():
                reconst_path = []
```

```python
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()

                print(f"Number of elements in final list:
                    ↪ {len(self.flatten_list(self.final_list))}")
                print('Path found: {}'.format(reconst_path))
                print(
                    f"My solution for N={N}: w={sum(len(_) for _ in
                        ↪ self.final_list)} (bloat={(sum(len(_) for _ in
                        ↪ self.final_list)-N)/N*100:.0f}%)"
                )
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                values = self.list_values[m]
                if m not in open_list and m not in closed_list:
                    # open_list.add(m)
                    open_list = self.insert_unique_element_into_list(open_list,
                        ↪ m)
                    # sort open_list by self.h
                    open_list = sorted(open_list, key=self.h)
                    parents[m] = n
                    g[m] = g[n] + weight

                else:
                    if g[m] + self.h(m) > g[n] + self.h(n) + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        # if m in closed_list:
                        #     closed_list.remove(m)
                        #     # open_list.add(m)
                        #     open_list =
                        ↪ self.insert_unique_element_into_list(open_list, m)
                        #     open_list = sorted(open_list, key=self.h)


            open_list.remove(n)
            open_list = sorted(open_list, key=self.h)
            closed_list = self.insert_unique_element_into_list(closed_list, n)

        print('Path does not exist!')
        return None
```

2.2 Results

2.3 Received Reviews

> Diego Mangasco
>
> ---
>
> REVIEW BY DIEGO GASCO (DIEGOMANGASCO) SET COVERING (GREEDY): I appreciated a lot the comparison between the professor's Naive greedy approach and your greedy approach! The idea to implement a sort of priority function to choose the best set to add to the solution is nice (a kind of cherry picking). I think you decided to take the set with lowest "f" because you want to keep low the total weight as you can. What if you merge this idea with the number of new elements that the new set can bring to your solution? You can try to find a sort of trade-off between having a new small set and having a new useful one!
>
> SET COVERING (A* TRAVERSAL USING PRIORITY QUEUE): In my implementation I basically used the same approach in developing my A* algorithm! Like you, I decided to implement my heuristics as the number of undiscovered elements, and I took as cost, the length of the new set added in the solution. I also noticed that, with cost sets as unit and not as the length of the new set, the process is much faster, but the solution that we reached is not optimal, so I decided to keep the length as cost.
>
> The only small difference with my implementation is the use of the data structures. To don't have to deal with list manipulation, I preferred to focused my structures in a more set-oriented way. But never mind, these are just personal preferences!
>
> SET COVERING (A* TRAVERSAL USING A FULLY CONNECTED GRAPH) Unfortunately I couldn't try this implementation of A*, because I didn't understand the data structure "adjacency list" and there isn't a block that starts this piece of code like for the previous solutions Reading your explanation about the algorithm idea, I can say that this approach can be useful with a solution space that is not huge, but can become computationally expansive with large N (due to the connections you might have to manage). But anyway with small/medium N it can be helpful in reducing the time of the classical A*.

Ramin

The code is written in a clear way and it's easy to understand. The code style is clear and the code is well organized in classes. The fact that you tried to implement a sort of priority function to choose the best set to add to the solution is nice and smart. Also you decided to implement your heuristics as the number of elements that have not been found yet, which is also a great idea. My only question is that , what is the best way to estimate the weight, considering the new items?

Arman

Hi Sid,

here is my review:

The algorithm you tried as an augmented greedy solution is finding good solutions for small Ns, e.g. 29 for N=20 which is close to the exact solution. (you forgot to put N=20 in the solutions as well, it's good to add it as you are using this as your baseline). The function which it uses for cost is actually a kind of heuristic used in a greedy context. It is an interesting use case. for large Ns, It does not improve the solution, although meaningfully reduces the number of visited nodes. It's a kind of behaviour we observe when using heuristics in other search algorithms as well.

for A* search, your code is pretty clean and organised specially implementing in a class which makes it reusable. the heuristic is reasonable and simple. comparing length as cost and unit cost is useful to see the difference. My experience was that not using cost and not keeping parents did not made much difference in this specific problem and it makes code much smaller and faster.

The fact that you used the itertools methods has made your code cleaner and more elegant. It is better to implement loops, e.g. in are_we_done() using comprehension, using inner loops in separate line will affect the speed significantly.

Using a fully connected graph is interesting experiment, I will follow.

Bests

2.4 Given Reviews

2.4.1 Shayan

Shayan's code

```python
import random
import logging
logging.getLogger().setLevel(logging.INFO)

def custom_search(N, seed):
    goal = set(range(N))
    covered = set()
    solution = list()
    all_lists = problem(N, seed=42)
    random.seed(seed)
    random.shuffle(all_lists) #shuffle list to pop random
    while goal != covered: #while set of covered nums is not equal to goal
        x = all_lists.pop(0) #pick a list from all_lists
        if not set(x) < covered: #if set of picked list is not a subset of
        ↪    covered
            solution.append(x) #append it to the solution
            covered |= set(x) #covered gets updated and becomes a union of
            ↪    covered plus picked set


    logging.info(
        f"custom search solution for N={N}: w={sum(len(_) for _ in solution)}
        ↪    (bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
    )
logging.getLogger().setLevel(logging.DEBUG)
for N in [5, 10, 20, 100, 500, 1000]:
    custom_search(N, 99)
```

Hi Shayan,

I had a look at your code and had a few thoughts:

1. You seem to be using a completely random approach to solving the problem, making a random, uninformed choice at each iteration of the loop. When running the algorithm with different random seeds, a different bloat factor and w are produced. The gist is that picking subsets randomly neither guarantees a heuristically optimal solution nor is the runtime optimised.

2. One suggestion to make informed decisions when choosing subsets is to sort the list by undiscovered elements / length of the list / other factors that affect the efficiency of the solution. This would still be a greedy, heuristically approximate solution that could improve both performance and runtime. Furthermore, you could consider traversing the list through more powerful search algorithms such as Djikstra or A-Star.

2. (Miscellaneous) While the results are in the notebook, perhaps you can add them to the markdown file to compare it with other algorithms in the future.

Thank you! If there are any other details I can add, please do let me know.

2.4.2 Arman

Arman's code

```python
import enum
from itertools import count
import logging
import random
from gx_utils import *
from heapq import heappush
from typing import Callable
import statistics
# import queues

logging.basicConfig(format="%(message)s", level=logging.INFO)


N = 1000
NUMBERS = {x for x in range(N)}



def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N //
        ↪  5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]

class State:
    def __init__(self, list_numbers:set):
        self.lists_ = list_numbers.copy()
    def add(self,item):
        self.lists_.add(item)
        return self
    def __hash__(self):
        #return hash(bytes(self.lists_))
        return hash(str(self.lists_))
    def __eq__(self, other):
        #return bytes(self.lists_) == bytes(other.lists_)
        return str(self.lists_) == str(other.lists_)
    def __lt__(self, other):
        #return bytes(self.lists_) < bytes(other.lists_)
        return str(self.lists_) < str(other.lists_)
    def __str__(self):
        return str(self.lists_)
```

```python
        def __repr__(self):
            return repr(self.lists_)
        def copy_data(self):
            return self.lists_.copy()
        def get_weight(self,ref_lists):
            return len([x for n in self.lists_ for x in ref_lists[n]])
        def get_items(self,ref_lists):
            return set([x for n in self.lists_ for x in ref_lists[n]])


    def goal_test(current_state:State,ref_lists):
        """get all the members of the lists in the current_state and check if it
        ↪    covers N"""

        current_numbers = {x for n in current_state.lists_ for x in ref_lists[n]}
        return current_numbers == NUMBERS

    def valid_actions(current_state:State,ref_lists):
        """returns set of indexes not currently added to this state"""
        return {indx for indx,_ in enumerate(ref_lists) if indx not in
        ↪    current_state.lists_}

    def result(current_state,action):
        next_state=State(current_state.copy_data()).add(action)
        return next_state

    def search(initial_state:State, ref_lists,priority_function:Callable):
        frontier = PriorityQueue()
        state = initial_state
        state_count = 0
        while state is not None and not goal_test(state,ref_lists):
            for a in valid_actions(state,ref_lists):
                new_state = result(state,a)
                if new_state not in frontier:
                    frontier.push(new_state,p=priority_function(new_state))
                elif new_state in frontier:
                    pass
            if frontier:
                state = frontier.pop()
                state_count+=1
            else:
                state = None

        logging.info(f"Found a solution with cost: {state.get_weight(ref_lists)}
        ↪    and {state_count} number of visited states, last state: {state}")

    def heuristic(state:State,ref_lists,N):
        remained = NUMBERS - state.get_items(ref_lists)
        return len(remained) + random.randint(0,len(remained)//2)
```

```
88
89     if __name__ == "__main__":
90         ref_lists = problem(N,seed=42)
91         #print(ref_lists)
92         initial_state = State(set())
93
94         # #Breath_first
95         # search(initial_state, ref_lists,priority_function=lambda state:
       ↪   state.get_weight(ref_lists))
96
97         # #Depth_first
98         # search(initial_state, ref_lists,priority_function=lambda state:
       ↪   -state.get_weight(ref_lists))
99
100        # #Heuristic
101        search(initial_state, ref_lists,priority_function=lambda state:
       ↪   heuristic(state,ref_lists, N))
```

Hi Arman,

Here are my observations with regard to your solution for Lab 1:

1. The priority queue is a suitable choice to store and select subsets in each iteration of your loop. All 4 traversal algorithms are compared by editing the priority function, and similar to mine, A-star performed best.

2. Your heuristic function is particularly interesting because it combines the "potential new elements" with a random number.

```
1   def heuristic(state:State,ref_lists,N):
2       remained = NUMBERS - state.get_items(ref_lists)
3       return len(remained) + random.randint(0,len(remained)//2)
```

There also wasn't an explanation in the Readme, so I'm very curious as to the reason behind this heuristic. I ran your code with and without this random component and found that using it improves performance for larger values of N such as $N = 100$ or $N = 500$, but not so for smaller values like $N = 20$. If you could add an explanation to your Readme about the heuristic, I would be very interested to read it.

3. Your algorithm does not hit a bottleneck for values of $N > 50$, in which case most people's code "exploded". Therefore, any solution, though not necessarily optimal, is reached.

4. One suggestion I have is to experiment with other heuristic functions, such as those that consider both the number of attainable new elements and the length of the incoming subset.

3 Lab 2

3.1 Solution

In this lab, we will take a GA approach to solving the set-covering problem. As a background, let's assume we have 500 potential lists that should form a complete subset.

The final product should be a list of 0s and 1s that indicate which lists should be included in the final set. We use a genetic approach to obtain this list via:

1. Mutation: randomly change a 0 to a 1 or vice versa

2. Crossover: randomly select a point in the list and swap the values after that point

3.1.1 Representing the problem

We will represent the problem as a list of 0s and 1s. The length of the list will be the number of lists we have. The 0s and 1s will indicate whether or not the list should be included in the final set.

The objective of the algorithm is to find an optimal (or at least as optimal as possible) set of 0s and 1s that will cover all the elements in the list.

3.1.2 Assessing Fitness

Based on knowledge obtained in previous labs, the heuristic function evolved and these were the factors I considered:

1. Potential duplicates

2. Undiscovered elements

3. Length of subset

The following equations were formulated for fitness assessment:

$$len(distinct\_elements) \tag{1}$$

$$len(distinct\_elements)/(num\_duplicates + 1) \tag{2}$$

$$len(distinct\_elements)/(num\_duplicates+1)-num\_undiscovered\_elements \tag{3}$$

N	W
5	.
10	10
20	24
50	100
100	197
500	1639
1000	3624

Table 1: Results of the algorithm

$$len(distinct\_elements)/(num\_undiscovered\_elements + 1) \qquad (4)$$

After multiple trials, the best fitness function is the simplest, which is simply the number of distinct elements.

3.2 Results

The results of the algorithm after 1000 generations (only the best results are reported) are shown in Table 1.

With larger values of N, a smaller population and offspring size is sufficient. Early stopping is used to detect the plateau, so the algorithm doesn't run endlessly. However, the minima is often reached in less than 100 generations.

3.2.1 The Case of Mutations

Plateau Detection and Dynamic Change of Mutation Rate Based on the rate of change of the fitness, the mutation rate (number of elements in genome to mutate) is adjusted.

```python
def choose_mutation_rate(fitness_log):
    # choose mutation rate based on change in fitness_log
    if len(fitness_log) == 0:
        return 0.2
    if len(fitness_log) < 3:
        considered_elements = len(fitness_log)
    else:
        considered_elements = 3
    growth_rate = np.mean(np.diff(fitness_log[-considered_elements:]))
    if growth_rate <= 0:
        return 0.4
    elif growth_rate < 0.5:
        return 0.3
    elif growth_rate < 1:
        return 0.01
```

```
16    else:
17        return 0.1
18
19 def plateau_detection(num_generations, fitness_log):
20     '''
21     Checks if the fitness has plateaued for the last num_generations.
22     '''
23     # this function is not used
24     return all(fitness_log[-num_generations] == fitness_log[-i] for i in range(1,
     ↪  num_generations))
```

3.3 Mutation Functions

3.3.1 Flip Mutation

```
1    def flip_mutation(genome, mutate_only_one_element=False):
2        '''
3        Flips random bit(s) in the genome.
4        Parameters:
5        mutate_only_one_element: If True, only one bit is flipped.
6        '''
7        modified_genome = genome.copy()
8        if mutate_only_one_element:
9            # flip a random bit
10           index = random.randint(0, len(modified_genome) - 1)
11           modified_genome[index] = 1 - modified_genome[index]
12       else:
13           # flip a random number of bits
14           num_to_flip = choose_mutation_rate(fitness_log) * len(modified_genome)
15           to_flip = random.sample(range(len(modified_genome)), int(num_to_flip))
16           # to_flip = random.sample(range(len(modified_genome)), random.randint(0,
     ↪  len(modified_genome)))
17           modified_genome = [1 - modified_genome[i] if i in to_flip else
     ↪  modified_genome[i] for i in range(len(modified_genome))]
18
19       # mutate only if it brings some benefit to the weight
20       # if calculate_weight(modified_genome) < calculate_weight(genome):
21       #     return modified_genome
22
23       return return_best_genome(modified_genome, genome)
```

3.3.2   Scramble Mutation

```python
def scramble_mutation(genome):
    '''
    Randomly scrambles the genome.
    '''
    # select start and end indices to scramble
    modified_genome = genome.copy()
    start = random.randint(0, len(modified_genome) - 1)
    end = random.randint(start, len(modified_genome) - 1)
    # scramble the elements
    modified_genome[start:end] = random.sample(modified_genome[start:end],
        len(modified_genome[start:end]))
    return return_best_genome(modified_genome, genome)
```

3.3.3   Swap Mutation

```python
def swap_mutation(genome):
    '''
    Randomly swaps two elements in the genome.
    '''
    modified_genome = genome.copy()
    index1 = random.randint(0, len(modified_genome) - 1)
    index2 = random.randint(0, len(modified_genome) - 1)
    modified_genome[index1], modified_genome[index2] = modified_genome[index2],
        modified_genome[index1]
    return return_best_genome(modified_genome, genome)
```

3.3.4   Inversion Mutation

```python
def inversion_mutation(genome):
    '''
    Randomly inverts the genome.
    '''
    modified_genome = genome.copy()
    # select start and end indices to invert
    start = random.randint(0, len(modified_genome) - 1)
    end = random.randint(start, len(modified_genome) - 1)
    # invert the elements
    modified_genome = modified_genome[:start] + modified_genome[start:end][::-1] +
        modified_genome[end:]
    return return_best_genome(modified_genome, genome)
```

3.4 Full Code

```python
import numpy as np
import itertools

def calculate_fitness(genome):
    '''
    Calculates the fitness of the given genome.
    The fitness is the number of unique elements
    The weight is the total number of elements in the genome
    '''
    # fitness is number of distinct elements in genome
    all_elements = []
    distinct_elements = set()
    weight = 0
    for subset, gene in zip(prob, genome):
        # if the particular element should be taken
        if gene == 1:
            distinct_elements.update(subset)
            weight += len(subset)
            all_elements += subset
    num_duplicates = len(all_elements) - len(set(all_elements))
    num_undiscovered_elements = len(set(range(N)) - distinct_elements)
    # print(set(range(N)) - distinct_elements)
    # print("num_undiscovered_elements", num_undiscovered_elements)
    # return num_undiscovered_elements, -weight
    # return len(distinct_elements), -weight
    # return num_undiscovered_elements / (len(distinct_elements) + 1), -weight
    return len(distinct_elements) / (num_undiscovered_elements + 1), -weight
    # other potential fitness functions:
    # return len(distinct_elements) / (num_duplicates + 1)
    # return len(distinct_elements) / (num_duplicates + 1) -
    #    num_undiscovered_elements, -weight
    # return len(distinct_elements) / (num_undiscovered_elements + 1), -weight

def generate_element():
    '''
    Randomly generates offspring made up of 0s and 1s.
    1 means the element is taken, 0 means it is not.
    '''
    genome = [random.randint(0, 1) for _ in range(N)]
    fitness = calculate_fitness(genome)
    # genome = np.random.choice([True, False], size=PROBLEM_SIZE)
    return Individual(genome, fitness)

initial_population = [generate_element() for _ in range(POPULATION_SIZE)]

len(initial_population)
```

```python
47   fitness_log = []
48
49   def calculate_weight(genome):
50       '''
51       Weight Function
52       Weight is the sum of the lengths of the subsets that are taken
53       '''
54       # select the subsets from prob based on the best individual
55       final = [prob[i] for i, gene in enumerate(genome) if gene == 1]
56       weight = len(list(itertools.chain.from_iterable(final)))
57       return weight
58
59   def choose_mutation_rate(fitness_log):
60       # choose mutation rate based on change in fitness_log
61       if len(fitness_log) == 0:
62           return 0.2
63       if len(fitness_log) < 3:
64           considered_elements = len(fitness_log)
65       else:
66           considered_elements = 3
67       growth_rate = np.mean(np.diff(fitness_log[-considered_elements:]))
68       if growth_rate <= 0:
69           return 0.4
70       elif growth_rate < 0.5:
71           return 0.3
72       elif growth_rate < 1:
73           return 0.01
74       else:
75           return 0.1
76
77   def plateau_detection(num_generations, fitness_log):
78       '''
79       Checks if the fitness has plateaued for the last num_generations.
80       '''
81       if len(fitness_log) < num_generations:
82           return False
83       return all(fitness_log[-num_generations] == fitness_log[-i] for i in range(1,
     ↪   num_generations))
84
85   def flip_mutation(genome, mutate_only_one_element=False):
86       '''
87       Flips random bit(s) in the genome.
88       Parameters:
89       mutate_only_one_element: If True, only one bit is flipped.
90       '''
91       modified_genome = genome.copy()
92       if mutate_only_one_element:
93           # flip a random bit
94           index = random.randint(0, len(modified_genome) - 1)
95           modified_genome[index] = 1 - modified_genome[index]
```

```python
96          else:
97              # flip a random number of bits
98              num_to_flip = choose_mutation_rate(fitness_log) * len(modified_genome)
99              to_flip = random.sample(range(len(modified_genome)), int(num_to_flip))
100             # to_flip = random.sample(range(len(modified_genome)), random.randint(0,
        ↪   len(modified_genome)))
101             modified_genome = [1 - modified_genome[i] if i in to_flip else
        ↪   modified_genome[i] for i in range(len(modified_genome))]
102
103     return modified_genome
104     # mutate only if it brings some benefit to the weight
105     # if calculate_weight(modified_genome) < calculate_weight(genome):
106     #     return modified_genome
107
108
109 def return_best_genome(genome1, genome2):
110     return genome1
111     # if calculate_fitness(genome1) > calculate_fitness(genome2):
112     #     return genome1
113     # else:
114     #     return genome2
115
116 def mutation(genome):
117     '''
118     Runs a randomly chosen mutation on the genome. Mutations are:
119     1. Bit Flip Mutation
120     2. Scramble Mutation
121     3. Swap Mutation
122     4. Inversion Mutation
123     Refer to README for more details.
124     '''
125     # check type of genome (debugging)
126     # if type(genome) == tuple:
127     #     print("genome is tuple")
128     #     print(genome)
129
130     possible_mutations = [flip_mutation, scramble_mutation, swap_mutation,
        ↪   inversion_mutation]
131     chosen_mutation = random.choice(possible_mutations)
132     return chosen_mutation(genome)
133
134     # if random.random() < 0.1:
135     #     for _ in range(num_elements_to_mutate):
136     #         index = random.randint(0, len(genome) - 1)
137     #         genome[index] = 1 - genome[index]
138     # mutate a random number of elements
139     # to_flip = random.randint(0, len(genome))
140     # # flip the bits
141     # return [1 - genome[i] if i < to_flip else genome[i] for i in
        ↪   range(len(genome))]
```

```python
142
143  def scramble_mutation(genome):
144      '''
145      Randomly scrambles the genome.
146      '''
147      # select start and end indices to scramble
148      modified_genome = genome.copy()
149      start = random.randint(0, len(modified_genome) - 1)
150      end = random.randint(start, len(modified_genome) - 1)
151      # scramble the elements
152      modified_genome[start:end] = random.sample(modified_genome[start:end],
         ↪  len(modified_genome[start:end]))
153      return return_best_genome(modified_genome, genome)
154
155  def swap_mutation(genome):
156      '''
157      Randomly swaps two elements in the genome.
158      '''
159      modified_genome = genome.copy()
160      index1 = random.randint(0, len(modified_genome) - 1)
161      index2 = random.randint(0, len(modified_genome) - 1)
162      modified_genome[index1], modified_genome[index2] = modified_genome[index2],
         ↪  modified_genome[index1]
163      return return_best_genome(modified_genome, genome)
164
165  def inversion_mutation(genome):
166      '''
167      Randomly inverts the genome.
168      '''
169      modified_genome = genome.copy()
170      # select start and end indices to invert
171      start = random.randint(0, len(modified_genome) - 1)
172      end = random.randint(start, len(modified_genome) - 1)
173      # invert the elements
174      modified_genome = modified_genome[:start] + modified_genome[start:end][::-1]
         ↪  + modified_genome[end:]
175      return return_best_genome(modified_genome, genome)
176
177  def crossover(genome1, genome2):
178      '''
179      Crossover the two genomes by randomly selecting a point
180      '''
181      # crossover at a random point
182      crossover_point = random.randint(0, len(genome1))
183      modified_genome = genome1[:crossover_point] + genome2[crossover_point:]
184      return modified_genome
185
186  def roulette_wheel_selection(population):
187      '''
188      Selects an individual from the population based on the fitness.
```

```python
189          '''
190          # calculate the total fitness of the population
191          total_fitness = sum([individual.fitness[0] for individual in population])
192          # select a random number between 0 and the total fitness
193          random_number = random.uniform(0, total_fitness)
194          # select the individual based on the random number
195          current_fitness = 0
196          for individual in population:
197              current_fitness += individual.fitness[0]
198              if current_fitness > random_number:
199                  return individual
200
201  def stochastic_universal_sampling(population):
202          '''
203          Select using Stochastic Universal Sampling.
204          '''
205          point_1 = random.uniform(0, 1)
206          point_2 = point_1 + 1
207          # In Progress
208
209  def rank_selection(population):
210          '''
211          Select using Rank Selection. Read more here:
212
    ↪   https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.h
213          '''
214          # sort the population based on the fitness
215          population.sort(key=lambda x: x.fitness[0], reverse=True)
216          # calculate the total rank
217          total_rank = sum([i for i in range(len(population))])
218          # select a random number between 0 and the total rank
219          random_number = random.uniform(0, total_rank)
220          # select the individual based on the random number
221          current_rank = 0
222          for i, individual in enumerate(population):
223              current_rank += i
224              if current_rank > random_number:
225                  return individual
226
227
228  def tournament(population, selection_method='tournament'):
229          '''
230          Selects the best individual from a random sample of the population.
231          '''
232          if selection_method == 'roulette':
233              participant = roulette_wheel_selection(population)
234              participant = Individual(participant.genome, participant.fitness)
235          elif selection_method == 'rank':
236              participant = rank_selection(population)
237              participant = Individual(participant.genome, participant.fitness)
```

```python
    else:
        participant = max(random.sample(population, k=2), key=lambda x:
        ↪  x.fitness)
        participant = Individual(participant.genome, participant.fitness)
    return participant

def generate(population, generation):
    '''
    Create offspring from the population using either:
    1. Cross Over + Mutation
    2. Mutation
    '''
    # can either cross over between two parents or mutate a single parent
    if random.random() < 0.2:
        parent = tournament(population)
        # if random.random() <= 0.3:
        #     genome = mutation(parent.genome)
        genome = mutation(parent.genome)
        child = Individual(parent, calculate_fitness(parent))
    else:
        # crossover
        parent1 = tournament(population)
        parent2 = tournament(population)
        genome = crossover(parent1.genome, parent2.genome)
        # if random.random() <= 0.3:
        #     genome = mutation(genome)
        genome = mutation(genome)
        child = Individual(genome, calculate_fitness(genome))

    fitness_log.append((generation + 1, child.fitness[0]))

    return child

best = max(initial_population, key=lambda x: x.fitness)

best_individual = max(initial_population, key=lambda x: x.fitness)
for i in range(NUM_GENERATIONS):
    # create offspring
    offspring = [generate(initial_population, i) for i in
    ↪  range(OFFSPRING_SIZE)]
    # calculate fitness
    # offspring = [Individual(child.genome, calculate_fitness(child.genome))
    ↪  for child in offspring]

    initial_population = initial_population + offspring
    initial_population = sorted(initial_population, key=lambda x: x.fitness,
    ↪  reverse=True)[:POPULATION_SIZE]

    fittest_offspring = max(initial_population, key=lambda x: x.fitness)
```

```
284        if fittest_offspring.fitness > best_individual.fitness:
285            best_individual = fittest_offspring
286
287    # get the best individual
288    print(calculate_weight(best_individual.genome))
```

3.5 Received Reviews

> **s295103**
>
> Your commitment to this lab can be seen from all the approaches you implemented and tested. My only issue is with the plateau detection function that is bound to always return False in that implementation. Also a suggestion: try to enforce the constraint that all individuals' genome must be a solution with full set cover; in this way you'll vastly reduce the search space.

s295103

Design considerations - Overall good solution, nice work trying multiple parent selection functions, different fitness functions, and using multiple mutation functions

Implementation considerations - After calling the problem() function it is necessary to reset the seed to a random value using 'random.seed()' otherways all runs will always use 42 as seed value, so they won't be truly random

```
1    def flip_mutation(genome, mutate_only_one_element=False): is never
     ↪ called with mutate_only_one_element=True
2    genome = mutation(parent.genome)
3    child = Individual(parent, calculate_fitness(parent))
4
```

should substituted by

```
1    genome = mutation(parent.genome)
2    child = Individual(genome, calculate_fitness(genome))
3
```

for the mutation to have effect, since in every mutation you do

```
1    def *_mutation(genome):
2        modified_genome = genome.copy()
3        ...
4        return modified_genome
```

```
1    initial_population = sorted(initial_population, key=lambda x:
     ↪ x.fitness, reverse=True)[:POPULATION_SIZE]
2    fittest_offspring = max(initial_population, key=lambda x: x.fitness)
```

can become

```
1    initial_population = sorted(initial_population, key=lambda x: x.fitness,
     ↪ reverse=True)[:POPULATION_SIZE]
2    fittest_offspring = initial_population[0]
```

so that you don't need to search for the max in the list you just sorted - The README and the important parts of the code are very clean and structured, but there are some comments, unused functions, an unfinished function, and other parts of the file that can be cleaned up a little

> Ricardo Nicida Kazama
>
> ----
>
> In the README, I was wondering if the function *return_best_genome*(*modified_genome*, genome) might disturb the exploration of your algorithm since a worse solution that could go towards the global optimum might be chosen instead of the current better solution that is going to a local optimum. Analyzing your code, I notice that the part where you would compare the genomes to pick the best is commented. Therefore, maybe you experienced what I previously mentioned. In the following part of the code, the use of the iterator "i" is a bit confusing since the one being taken into account for the function generate(*initial_population*, i) is the one in range(*OFFSPRING_SIZE*). However, from what I understood, the second input should be the generation number.
>
> ```python
> for i in range(NUM_GENERATIONS):
>     # create offspring
>     offspring = [generate(initial_population, i) for i in
>     range(OFFSPRING_SIZE)]
> ```
>
> Highlights/overall: The solution includes many different mutations which show an extra effort to improve the results with a broad approach. The change in the mutation rate based on the *fitness_log* is an interesting idea and seems to be effective. The code and results are very good!

3.6 Given Reviews

3.6.1 Erik

Erik's code

```python
# Should be used to init solution space, return a list of list
def select_rand_solution(full_input):
    population = []
    random.seed(None)
    for i in range(POPULATION_SIZE):
        population.append(random.sample(full_input, random.randint(1,
        len(full_input))))
    return population


# check if one solution is valid
def goal_check(curr):
    curr = [item for sublist in curr for item in sublist]
    return set(curr) == set(range(N))
```

```python
14
15
16   def fitness_function(entry, goal_set):
17       duplicates = len(entry) - len(set(tuple(entry)))
18       miss = len(goal_set.difference(set(entry)))
19       return (-1000 * miss) - duplicates
20
21
22   def calculate_fitness(individual):
23       flat_individual = [item for sublist in individual for item in sublist]
24       fitness_val = fitness_function(flat_individual, set(range(N)))
25       return fitness_val
26
27
28   def select_parents(population):
29       nr_of_boxes = int(POPULATION_SIZE * (POPULATION_SIZE + 1) / 2)
30       random.seed(None)
31       random_wheel_nr = random.randint(1, nr_of_boxes)
32       parent_number = POPULATION_SIZE
33       increment = POPULATION_SIZE - 1
34       curr_parent = 0
35       while random_wheel_nr > parent_number:
36           curr_parent += 1
37           parent_number += increment
38           increment -= 1
39       return population[curr_parent]
40
41
42   # randomize an index and merge 0-index from parent 1 and index-len of parent two,
     ↪  mutate with 5% chance
43   def crossover(first_parent, second_parent):
44       slice_index_one = random.randint(0, min(len(first_parent[0]) - 1,
         ↪  len(second_parent[0]) - 1))
45       child = first_parent[0][:slice_index_one] +
         ↪  second_parent[0][slice_index_one:]
46       return child
47
48
49   # mutate child and return
50   def mutate_child(individual, problem_space):
51       index = random.randint(0, len(individual) - 1)
52       random_list = problem_space[random.randint(0, len(problem_space) - 1)]
53       random_gene = random_list[random.randint(0, len(random_list) - 1)]
54       individual = individual[:index] + individual[index+1:] + [random_gene]
55       return individual
56
57
58   def update_population(population, new_children):
59       new_population = population + new_children
60       sorted_population = sorted(new_population, key=lambda i: i[1], reverse=True)
```

```python
61        return sorted_population[:POPULATION_SIZE]
62
63
64  def main():
65      logging.basicConfig(level=logging.DEBUG)
66      problem_space = problem(N, seed=42)
67      population = select_rand_solution(problem_space)
68
69      # should hold current population with the calculated fitness
70      current_individuals = []
71
72      # setup data structure, list of tuples containing ([entries], fitness) and
73      for individual in population:
74          current_individuals.append((individual, calculate_fitness(individual)))
75
76      current_individuals = sorted(current_individuals, key=lambda l: l[1],
77          reverse=True)
78      counter = 0
79      while counter < NR_OF_GENERATIONS:
80          # a) Select individuals with a good fitness score for reproduction.
81          cross_over_list = []
82          for i in range(OFFSPRING_SIZE):
83              parent_one = select_parents(current_individuals)
84              parent_two = select_parents(current_individuals)
85
86              # b) Let them produce offspring. Mutate with 5% chance
87              tmp_child = crossover(parent_one, parent_two)
88              if random.random() > 0.95:
89                  tmp_child = mutate_child(tmp_child, population)
90
91              cross_over_list.append((tmp_child, calculate_fitness(tmp_child)))
92
93          current_individuals = update_population(current_individuals,
94              cross_over_list)
95          counter += 1
96
97      for solution in current_individuals:
98          if goal_check(solution[0]):
99              logging.info(f'Best solution for N={N} was
                  {current_individuals[0][0]} \nWith a weight of {sum(len(_) for _
                  in current_individuals[0][0])}')
              break
```

Hi Eric,

Here's my review concerning your approach to lab 2.

There are a few high-level, cosmetic attributes you did well: 1. Each function is well-documented and well-labelled, so I could easily understand the purpose of each one. One way to improve could be to leverage Python docstrings, where you

can also explain input parameters and output values. To do this, add:

```python
def mutation(genome):
    '''
    Function mutates genome using .... strategy, etc.
    args:
    genome: str - Input genome
    '''
```

3. Using a Python script made it easy for me to run code iteratively for many different values of N/Offspring sizes/etc. without having to run all the cells. I was able to reproduce your best results after a few tries.

Let's break down the solution itself:

1. I noticed that you leveraged a completely random roulette-wheel-based selection, which leverages completely on random chance, compared to a fitness-based tournament selection which performed better (at least from my experience with this lab). Perhaps, you could try experimenting with different parent selection methods instead of just one.

2. Your fitness function is particularly interesting, standing out from most others I've seen. It takes into account duplicates in the subset:

```python
def fitness_function(entry, goal_set):
    duplicates = len(entry) - len(set(tuple(entry)))
    miss = len(goal_set.difference(set(entry)))
    return (-1000 * miss) - duplicates
```

I understand that the infinitesimal blowup by $*1000$ may theoretically help punish the algorithm if it is far from the goal. I modified your code with 2 different fitness functions:

```python
return miss-duplicates
```

```python
return (-1000 * miss)-duplicates
```

and the results were the same, so I look forward to reading about your motivation for this in the README.

Since you're only subtracting the two values (one is much larger than the other), you can do 1 of 2 things to improve convergence: divide the values, or return them as a tuple (like we did for the first lab). You could also try different mathematical equations for the fitness function, that takes into account duplicates, undiscovered

elements, length, etc., kind of like the heuristic functions we used early for graph algorithms.

3. Only one type of mutation is used (randomly flipping a bit). You could try other mutation methods and randomly choose between them to increase exploration power.

4. The probability to decide whether to mutate is quite high. In the Telegram chat, most people reported that mutations were detrimental to reaching minima, so I understand why you might have limited your mutations, but perhaps you could vary this number based on the changing fitness. Perhaps, mutate more often/more extensively to explore and reduce the vigour to exploit. You can also experiment with permutations of evolution like recombination + mutation, recombination only, mutation only, etc. All these contribute to the exploration power of your approach.

5. There is definitely a scaling problem for large values of N, such as $N = 1000$. One thing to note is that minima is often reached within a fraction of 1000 generations (I logged your generational results out).

5. Representing the problem space as 0s and 1s could result in cleaner code and faster computation, but this is more of a personal preference and does not really affect the solution.

All in all, good job! I just want to read more about your exciting fitness function. Let's discuss below!

3.6.2 Karl

Karl's code

```python
# helping functions

def lists_to_set(genome):
    """
    convert genome to set
    :param genome: the sub-lists with random integers between 0 and N-1
    :return: set of contained elements in the genome
    """
    list_elems = [single_elem for l in genome for single_elem in l]
    s = set(list_elems)
    return s

# find out how many duplicates there are in the population
def count_duplicates(genome):
    """
    Count how many duplicates there are in the genome
    :param genome: the sub-lists with random integers between 0 and N-1
```

```
18          :return: the count
19          """
20          list_elems = [single_elem for l in genome for single_elem in l]
21          duplicates = sum([len(list(group))-1 for key, group in
            ↪   groupby(sorted(list_elems))])
22          return duplicates
23      # to initialize the population
24      def create_population(STATE_SPACE, GOAL):
25          """
26          Initialize the population.
27          :param STATE_SPACE: List of lists generated from problem-function
28          :param GOAL: set of integers from 0 to N-1
29          :return: a list of tuples: (genome,fitness), for each individual in the
        ↪   population.
30          """
31          population = []
32          for _ in range(POPULATION_SIZE):
33              individual = []
34              for _ in range(random.randint(1,len(STATE_SPACE))):
35                  l = random.choice(STATE_SPACE)
36                  if l not in individual: #check duplicates here
37                      individual.append(l)
38              #individual =
                ↪   random.choices(STATE_SPACE,k=random.randint(1,len(STATE_SPACE)))
39              fitness = compute_fitness(individual, GOAL)
40              population.append((individual,fitness))
41          return population
42
43      def compute_fitness(genome, GOAL):
44          """
45          fitness is a tuple of (-#of_elems_missing,-#duplicates) which should be
        ↪   maximized
46          :param genome: the sub-lists with random integers between 0 and N-1
47          :param GOAL: set of integers from 0 to N-1
48          :return: the fitness
49          """
50          # violated constraints, i.e. how many elements are missing
51          vc = GOAL.difference(lists_to_set(genome))
52          duplicates = count_duplicates(genome)
53          # it is worse to lack elements than having duplicates
54          fitness = (-len(vc), -duplicates)
55          return fitness
56
57      def goal_check(genome, GOAL):
58          """
59          Check if all required elements are in the genome
60          :param genome: the sub-lists with random integers between 0 and N-1
61          :param GOAL: set of integers from 0 to N-1
62          :return: boolean value if goal reached or not
63          """
```

```python
64      return GOAL==lists_to_set(genome)


66  def parent_selection(population):
67      """
68      parent selection using ranking system
69      P(choose fittest parent) = POPULATION_SIZE/n_slots
70      P(choose second fittest parent) = (POPULATION_SIZE-1)/n_slots
71      ...
72      P(choose least fit parent) = 1/n_slots
73      :param population: list of individuals
74      :return: parent to generate offspring
75      """
76      ranked_population = sorted(population, key=lambda t : t[1], reverse=True)
77      # number of slots in spinning wheel = POPULATION_SIZE(POPULATION_SIZE+1)/2
        ↪   (arithmetic sum)
78      n_slots = POPULATION_SIZE*(POPULATION_SIZE+1)/2
79      wheel_number = random.randint(1,n_slots)
80      curr_parent = 0
81      parent_number = POPULATION_SIZE
82      increment = POPULATION_SIZE-1
83      while wheel_number > parent_number:
84          curr_parent +=1
85          parent_number +=increment
86          increment -= 1
87      return ranked_population[curr_parent]


89  # make one child from each cross-over, and mutate with low prob
90  def cross_over(parent1, parent2, STATE_SPACE, mutation_prob = 0.1):
91      """
92      Compute cross-over between two selected parents. Mutate child with
    ↪   mutation_prob.
93      :param parent1: individual
94      :param parent2: individual
95      :param STATE_SPACE: List of lists generated from problem-function
96      :param mutation_prob: the probability to perform mutation
97      :return: the child created
98      """
99      cut1 = random.randint(0,len(parent1[0]))
100     cut2 = random.randint(0,len(parent2[0]))
101     child = parent1[0][:cut1]+parent2[0][cut2:]
102     if random.random() < mutation_prob:
103         mutate(child, STATE_SPACE)
104     return child



107 def mutate(child, STATE_SPACE):
108     """
109     Replace one list in the child with a random one from the state space.
110     :param child:
111     :param STATE_SPACE:
```

```python
112         :return: the mutated child
113         """
114         idx = random.randint(0,len(child))
115         #child = child[:idx] + child[idx+1:] +
       ↪   STATE_SPACE[random.randint(0,len(STATE_SPACE)-1)]
116         i = 0
117         while i<10:
118             i+=1
119             if STATE_SPACE[random.randint(0,len(STATE_SPACE)-1)] not in child:
120                 child = child[:idx] + child[idx+1:] +
                 ↪   STATE_SPACE[random.randint(0,len(STATE_SPACE)-1)]
121                 break
122         return child
123
124     def update_population_plus(population, offspring):
125         """
126         Using the plus strategy to update population to next generation.
127         :param population:
128         :param offspring:
129         :return: the best individuals in union(population, offspring)
130         """
131         tot = population + offspring
132         ranked_population = sorted(tot, key=lambda t : t[1], reverse=True)
133         return ranked_population[:POPULATION_SIZE]
134
135     def update_population_comma(offspring):
136         """
137         Using the plus strategy to update population to next generation.
138         :param offspring:
139         :return: the best individuals in from offspring
140         """
141         ranked_pop = sorted(offspring, key=lambda t : t[1], reverse=True)
142         return ranked_pop[:POPULATION_SIZE]
143
144     def update_mutation_prob(best_solution, best_this_iter, mutation_param, it):
145         """
146         Update the mutation probability according to how the performance evolves. If
       ↪   no improvement, mutation probability increases (favour exploration). If
       ↪   improvement, mutation probability decreases (favour exploitation).
147         :param best_solution: The best solution so far
148         :param best_this_iter: The best solution of this generation
149         :param mutation_param:
150         :param it: iteration number
151         :return: the new mutation probability
152         """
153         if best_solution[1] >= best_this_iter[1]:
154             mutation_param +=1
155         elif best_solution[1] >= best_this_iter[1] and mutation_param>0:
156             mutation_param -= 1
157         return mutation_param/(1+it), mutation_param
```

```python
158  def solve_problem(N):
159      STATE_SPACE = problem(N,seed=42)
160      GOAL = set(range(N))
161      population = create_population(STATE_SPACE, GOAL)
162      best_sol = population[0] #to be updated after each iter
163      found_in_iter = 0 #to be updated
164      mutation_param = 1 #increase if solution doesn't improve
165      mutation_prob = 0.1 #init value
166      for i in range(ITERS):
167          offspring = []
168          for __ in range(OFFSPRING_SIZE):
169              parent1, parent2 = parent_selection(population),
                 ↪ parent_selection(population)
170              child = cross_over(parent1,parent2, STATE_SPACE, mutation_prob)
171              child_fitness = compute_fitness(child, GOAL)
172              offspring.append((child,child_fitness))
173          population = update_population_plus(population, offspring)
174          #population = update_population_comma(offspring)
175          best_curr = sorted(population, key=lambda l:l[1], reverse=True)[0]
176          mutation_prob, mutation_param = update_mutation_prob(best_sol, best_curr,
                 ↪ mutation_param, i)
177          if goal_check(best_curr[0],GOAL) and best_curr[1] > best_sol[1]:
178              best_sol = best_curr
179              found_in_iter = i
180      logging.info(f'Best solution found in {found_in_iter} iters and has weight
             ↪ {-best_sol[1][1]}')
181      return best_sol
182  # main
183
184  # settings
185  POPULATION_SIZE = 50
186  OFFSPRING_SIZE = 30
187  ITERS = 100
188
189  for N in [5,10,20,50,100,1000,2000]:
190      best_sol = solve_problem(N)
191      print(f'N = {N}')
192      logging.info(f'The best weight for N = {N}: {-best_sol[1][1]+N}')
```

Hi Karl,

Here's my review about your approach to lab 2. The key positives (cosmetic and logical):

1. The notebook is well-documented and cells are used appropriately. I also like that you described the steps of the algorithm before implementing it.

2. You were the only other person who compared both the (parent, offspring) and (parent + offspring) method for the algorithm. As evident in the results, parent + offspring produced more optimal weights for smaller values of N.

3. Parent selection also accounts for the second and third-best genomes, which

could add more diversity to the selection algorithm. I don't fully understand how your wheel selection works and would love to read more about this either through comments/README.

Potential Improvements:

1. Your fitness function also includes duplicates, which can be detrimental to the optimality of any solution, and using a tuple is a good idea. You could also try different mathematical heuristic-like combinations of these various factors, like subtracting/dividing.

```
# it is worse to lack elements than having duplicates
fitness = (-len(vc), -duplicates)
return fitness
```

2. Only one type of mutation is used, so you could try multiple different mutation methods and randomly choose between them. Specific methods are more aggressive than others, so the choice between methods could also be based on fitness improvement.

4. The mutation probability is constant, and could potentially be dynamic, with the same intuition behind (2) above. In cases where the fitness is worsening, you could mutate more aggressively, and when it's time to exploit, it could be reduced as a solution is nearing.

```
def cross_over(parent1, parent2, STATE_SPACE):
    cut1 = random.randint(0,len(parent1[0]))
    cut2 = random.randint(0,len(parent2[0]))
    child = parent1[0][:cut1]+parent2[0][cut2:]
    # dynamic_threshold = do some computation here to derive probability from the
    ↪   change in fitness
    # if random.random() < dynamic_threshold
        mutate(child, STATE_SPACE)
    return child
```

6. You could experiment with different combinations of crossover and mutation, based on different probabilities instead of simply crossover followed by mutation. Certain evolution methods are more aggressive than others, so this could mix it up a bit.

All in all, good job!

3.6.3 Ricardo

Ricardo's code

```python
from itertools import compress
from collections import namedtuple
N = 5
POPULATION_SIZE = 10
OFFSPRING_SIZE = 2
GENERATIONS = 5
PROB = 0.5 # probability to choose 1 for each one of the locus in the
    population
Individual = namedtuple('Individual', ('genome', 'fitness','goal_reached',
    'w'))
# this function evaluats the fitness and if the goal was reached
def fitness_goal_eval(list_of_lists, genome, goal):
    current_goal = goal
    solution = list(compress(list_of_lists, genome))
    # fitness = 0
    new_elements = 0
    repeated_elements = 0
    w = 0
    goal_reached = False

    if len(solution) == 0:
        return 0, False, 0

    for list_ in solution:
        list_length = len(list_)
        list_ = set(list_)
        cg_length = len(current_goal)
        current_goal = current_goal - list_
        cg_new_length = len(current_goal)

        # fitness += cg_length - cg_new_length   # new elements (positive)
        # fitness += (cg_length - cg_new_length) - list_length # repeated
            elements (negative)
        new_elements += cg_length - cg_new_length   # new elements
        repeated_elements += list_length - (cg_length - cg_new_length) #
            repeated elements

        w += list_length

    if cg_new_length == 0:
        goal_reached = True

    fitness = new_elements - repeated_elements

    return fitness, goal_reached, w


def generate_population(list_of_lists, goal):
    population = list()
```

```python
47        genomes = [tuple(random.choices([1, 0], weights=(PROB,1-PROB),
          ↪ k=len(list_of_lists))) for _ in range(POPULATION_SIZE)]
48
49        for genome in genomes:
50            fitness, goal_reached, w = fitness_goal_eval(list_of_lists, genome,
              ↪ goal)
51            population.append(Individual(genome, fitness, goal_reached, w))
52        return population
53
54
55    def select_parent(population, tournament_size=2):
56        subset = random.choices(population, k=tournament_size)
57        return max(subset, key=lambda i: i.fitness)
58
59
60    def cross_over(p1, p2, genome_size, list_of_lists, goal):
61        g1, f1 = p1.genome, p1.fitness
62        g2, f2 = p2.genome, p2.fitness
63        cut = int((f1+1e-6)/(f1+f2+1e-6)*genome_size)    # the cut is proportional
          ↪ to the fitness of the genome
64        ng1 = g1[:cut] + g2[cut:]
65        return ng1
66
67
68    def mutation(g, genome_size, k=1):  # for larger N try to eliminate some of the
      ↪ 1 in the genome because the bloat was getting to high
69        for _ in range(k):
70            cut = random.randint(1, genome_size)
71            if N < 20:
72                ng = g[:cut-1] + (1-g[cut-1],) + g[cut:]
73            elif N< 500:
74                cut_size = int(genome_size*0.2)
75                new_genome_cut = tuple(random.choices([1, 0], weights=(1, 39),
                  ↪ k=2*cut_size))
76                ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
77            else:
78                cut_size = int(genome_size*0.2)
79                new_genome_cut = tuple(random.choices([1, 0], weights=(1, 99),
                  ↪ k=2*cut_size))
80                ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
81        return ng
82    def genetic_algorithm():
83        # create problem
84        list_of_lists = problem(N, seed=42)
85        genome_size = len(list_of_lists)
86        goal = set(range(N))
87
88        # create the population
89        population = generate_population(list_of_lists, goal)
90
```

```python
    for g in range(GENERATIONS):
        population = sorted(population, key=lambda i: i.fitness,
         ↪  reverse=True)[:POPULATION_SIZE-OFFSPRING_SIZE]

        for i in range(OFFSPRING_SIZE):
            p1 = select_parent(population,
             ↪  tournament_size=int(0.2*genome_size))
            p2 = select_parent(population,
             ↪  tournament_size=int(0.2*genome_size))
            o = cross_over(p1, p2, genome_size, list_of_lists, goal)
            fitness, goal_reached, w = fitness_goal_eval(list_of_lists, o,
             ↪  goal)
            o = mutation(o, genome_size, k=2)

            population.append(Individual(o, fitness, goal_reached, w))



    for i in population:
        if i.goal_reached:
            return i, population

    print(f"No solution for current population (N={N})")
    return None, population
N = 500
POPULATION_SIZE = 100
OFFSPRING_SIZE = 20
GENERATIONS = 200
PROB = 0.5

logging.getLogger().setLevel(logging.INFO)

solution, population = genetic_algorithm()
if solution != None:
    logging.info(
        f" Genetic algorithm solution for N={N:,}: "
        + f"fitness={solution.fitness:,} "
        + f"w={solution.w:,} "
        + f"(bloat={solution.w/N*100:.0f}%)"
    )
INFO:root: Genetic algorithm solution for N=500: fitness=-1,980 w=2,980
↪  (bloat=596%)
POPULATION_SIZE = 50
OFFSPRING_SIZE = 20
GENERATIONS = 200
PROB = 0.5

logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20, 100, 500, 1000]:
```

```
136        solution, population = genetic_algorithm()
137        if solution != None:
138            logging.info(
139                f" Genetic algorithm solution for N={N:,}: "
140                + f"fitness={solution.fitness:,} "
141                + f"w={solution.w:,} "
142                + f"(bloat={solution.w/N*100:.0f}%)"
143            )
```

Hi Ricardo,

Here is my review pertaining to your approach to Lab 2.

Positives (both cosmetic and logical):

1. Your dynamic mutation method where you changed the strategy for different values of N is quite interesting. Larger N values will have 1s removed more aggressively, which is quite intuitive. Though this is not completely "dynamic", it is a good start. Just like your crossover is proportional to fitness, the same could be done for the "aggression" of the mutation.

```
1        if N < 20:
2            ng = g[:cut-1] + (1-g[cut-1],) + g[cut:]
3        elif N< 500:
4            cut_size = int(genome_size*0.2)
5            new_genome_cut = tuple(random.choices([1, 0], weights=(1, 39),
             ↪  k=2*cut_size))
6            ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
7        else:
8            cut_size = int(genome_size*0.2)
9            new_genome_cut = tuple(random.choices([1, 0], weights=(1, 99),
             ↪  k=2*cut_size))
10           ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
```

> A quick tip: both the 'elif' and 'else' have the same code block, so it could just be an 'if' an 'else'.

2. The tournament size dynamically changes based on the genome size. Yuri et al. (2018) advocated against the indiscriminate tournament size of $k = 2$.

3. The fitness function seems to be heuristic-like, considering both the number of new and repeated elements.

4. You used a list of 0s and 1s as binary indicators of whether to take a list in the subset. I feel that this is an efficient and intuitive representation.

5. You added an extra attribute 'goal_reached' to each element of the population, so when you loop through to find the final solution at the end, you not only get a working solution, but the one which produces the highest fitness.

Things to look at:

1. A mutation of some form is *always* applied in each generation after

crossover. To balance between exploitation and exploration, you could choose to mutate based on a random probability/change of the fitness function. I personally found that aggressive mutations worked well in early generations, but as minima is nearing, continually mutating did not improve the solution. One option is to choose between (i) crossover only, (ii) crossover then mutate, (iii) mutate only, etc. in each generation.

```python
if random.random() < threshold or some_fitness_based_condition:
            # crossover
elif random.random() < threshold:
            # crossover + mutate
elif ....:
             # mutate
```

2. MINOR- Reporting results in a table in the README makes it easier to compare.

All in all, good job!

3.6.4 Francesco

Francesco's code

```python
import random
import logging
import numpy as np
from collections import namedtuple
def problem(N, seed=None):
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5,
        ↪  N // 2))))
        for n in range(random.randint(N, N * 5))
    ]
def tournament(population, tournament_size=2):
    return max(random.choices(population, k=tournament_size), key=lambda i:
    ↪  i.fitness)

def w(genome):
    return sum(len(_) for _ in genome)

def covering(genome):
    s = set()
    for _ in genome:
        s =  s.union(set(_))
    return len(s)

def intersection(lst1, lst2):
```

```python
        lst3 = [value for value in lst1 if value in lst2]
        return lst3

    def shuffle(g1,g2,g3):
        a = [l for l in g1 if l not in g3]
        b = [l for l in g2 if l not in g3]
        gnew = g3.copy()

        if a:
            c = 1
        else:
            c = 0
        for i in range(max(len(a),len(b))):
            if c :
                if a and i < len(a):
                    gnew.append(a[i])
                if b:
                    c = 0

            else:
                if b and i < len(b):
                    gnew.append(b[i])
                if a:
                    c = 1

        return gnew

    def cross_over(g1, g2):
        g3 = intersection(g1,g2)
        g3 = shuffle(g1,g2,g3)
        return g3


    def mutation(genome):

        mutation = random.choice(all_lists)
        if mutation in genome:
            genome.remove(mutation)
        else:
            genome.append(mutation)

        return genome

    def create_population(mu):
        population = []
        for i in range(mu):
            g = []
            while covering(g) != N:
                if len(g) < N*2:
                    r = random.choice(all_lists)
```

```
74                    if r not in g:
75                        g.append(r)
76                else:
77                    g = []
78            population.append(g)
79        return [Individual(g, tuple((covering(g),-w(g)))) for g in population]
80    N = 1000
81    all_lists = problem(N,seed=42)
82    Individual = namedtuple("Individual", ["genome", "fitness"])
83    mu = 2000
84    GENERATIONS = 100
85    OFFSPRINGS_SIZE = 1100
86    population = create_population(mu)
87
88    for g in range(GENERATIONS):
89        new_population = []
90        for _ in range(OFFSPRINGS_SIZE):
91            o = []
92            if random.random() < 0.001:
93                p = tournament(population)
94                o = mutation(p.genome)
95            else:
96                p1 = tournament(population)
97                p2 = tournament(population)
98                o = cross_over(p1.genome, p2.genome)
99            new_population.append(Individual(o, tuple((covering(o),-w(o)))))
100       population += new_population
101       population = sorted(population, key= lambda i : i.fitness,
       ↪   reverse=True)[:mu]
102
103   print(f'w={w(population[0].genome)}, cov={covering(population[0].genome)}')
```

Hi Francesco,

Here is my quick review pertaining to your approach to Lab 2.

Positives (both cosmetic and logical):

1. The README was well-documented and I was able to come close to your best results when running the notebook locally with the specified hyperparameters.

2. The shuffling after the intersection seems to add a sort of random diversity to the evolved set, so that is great. I'll take inspiration from this. However, I don't fully understand the mechanism of the shuffle function. It would be great if I could read some comments or if the variables a, b and c could be renamed.

3. The hyperparameters like offspring size were varied for different sizes of N, which was the same thing I did. I was wondering if there was an intuition for choosing certain values. This could be explained in the README.

Some things to look at:

1. Mutations are rarely applied in each generation (at an extremely low probability of 0.001). I recall there was a discussion on the Telegram group about the detrimental effect mutating had on the final solution, so I understand why you might have done this. However, I found that mutating in early generations helps improve exploration power.

2. A constant 'tournament_size' of 2 is used for all values of N. Although early papers suggested the use of a constant, indiscriminate tournament size, recent papers like Yuri et al. advocated for adapting this parameter. I also used a constant size in my work, but this is something we can look at.

3. In the instances where mutation is done, only one type of mutation is used. You could try a diverse mix of mutation strategies like flipping, inversion, scrambling, etc. Since mutations haven't worked too well for you so far, the choice of strategy and aggression could be something to explore.

4. Runtime is rather slow for large values of N, which was the same case for me. This could also be because of the large number of generations (2000) the solution has to iterate through.

All in all, good job.

4 Lab 3

Nim is a simple game where two players take turns removing objects from a pile. The player who removes the last object wins. The game is described in detail here. There is a mathematical strategy to win Nim, by ensuring you always leave the opponent with a nim-sum number of objects (groups of 1, 2 and 4).

In this notebook, we will play nim-sum using the following agents:

1. An agent using fixed rules based on nim-sum

2. An agent using evolved rules

3. An agent using minmax

4. An agent using reinforcement learning (both temporal difference learning and monte carlo learning)

4.1 Solution

4.1.1 Fixed Rules

I came up with multiple rules, through discussion with friends and through research papers that define fixed rules for playing Nim. There are currently 4 rules implemented. The rules are as follows:

1. If one pile, take x number of sticks from the pile.

2. If two piles, take x number of sticks from the larger pile.

3. If two piles: a. If 1 pile has 1 stick, take x sticks b. If 2 piles have multiple sticks, take x sticks from the larger pile

4. If three piles and two piles have the same size, remove all sticks from the smallest pile

5. If n piles and n-1 piles have the same size, remove x sticks from the smallest pile until it is the same size as the other piles

Approach 1: A Lot of If-Elses The above rules are applied directly. An if-else sequence decides which strategy to employ based on the current layout and statistics on the nim board.
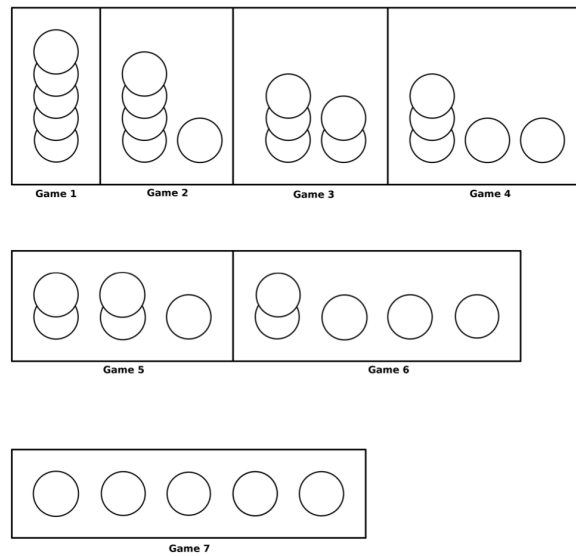
Player 1 has a winning strategy for all of these games! In game 1, the first player can just take all of the stones immediately. In games 2, 3, 4, and 5, the first player should use his first move to leave his opponent with two piles of the same size, and then mirror the opponents moves for the rest of the game (this will be explained in more detail in exercise 4). In games 6 and 7, the first player should use his first move to leave his opponent with four piles with one stone each; since they each can only take one stone for each of the next four turns, player 1 will win. □

Figure 2: Fixed Rules

```python
from collections import Counter
from copy import deepcopy
from itertools import accumulate
import logging
from operator import xor
import random
from typing import Callable

from lib import Genome, Nim, Nimply


class FixedRuleNim:
    def __init__(self):
        self.num_moves = 0
        self.OFFSPRING_SIZE = 30
        self.POPULATION_SIZE = 100
        self.GENERATIONS = 100
        self.nim_size = 5

    def nim_sum(self, nim: Nim):
        '''
        Returns the nim sum of the current game board
        by taking an XOR of all the rows.
        Ideally, agent should try to leave nim sum of 0 at the end of turn
        '''
        *_, result = accumulate(nim.rows, xor)
        return result
```

```python
        def init_population(self, population_size, nim: Nim):
            '''
            Initialize population of genomes,
            key is rule, value is number of sticks to take
            The rules currently are:
            1. If one pile, take $x$ number of sticks from the pile.
            2. If two piles:
                a. If 1 pile has 1 stick, wipe out the pile
                b. If 2 piles have multiple sticks, take x sticks from any pile
            3. If three piles and two piles have the same size, remove all sticks
    ↪    from the smallest pile
            4. If n piles and n-1 piles have the same size, remove x sticks from
    ↪    the smallest pile until it is the same size as the other piles
            '''
            population = []
            for i in range(population_size):
                # rules 3 and 4 are fixed (apply for 3 or more piles)
                # different strategies for different rules (situations on the
                ↪    board)
                individual = {
                    'rule_1': [0, random.randint(0, (nim.num_rows - 1) * 2)],
                    'rule_2a': [random.randint(0, 1), random.randint(0,
                    ↪    (nim.num_rows - 1) * 2)],
                    'rule_2b': [random.randint(0, 1), random.randint(0,
                    ↪    (nim.num_rows - 1) * 2)],
                    'rule_3': [nim.rows.index(min(nim.rows)), min(nim.rows)],
                    'rule_4': [nim.rows.index(max(nim.rows)), max(nim.rows) -
                    ↪    min(nim.rows)]
                }
                genome = Genome(individual)
                population.append(genome)
            return population

        def statistics(self, nim: Nim):
            '''
            Similar to Squillero's cooked function to get possible moves
            and statistics on Nim board
            '''
            # logging.info('In statistics')
            # logging.info(nim.rows)
            stats = {
                'possible_moves': [(r, o) for r, c in enumerate(nim.rows) for o
                ↪    in range(1, c + 1) if nim.k is None or o <= nim.k],
                # 'possible_moves': [(row, num_objects) for row in
                ↪    range(nim.num_rows) for num_objects in range(1,
                ↪    nim.rows[row]+1)],
                'num_active_rows': sum(o > 0 for o in nim.rows),
                'shortest_row': min((x for x in enumerate(nim.rows) if x[1] > 0),
                ↪    key=lambda y: y[1])[0],
```

```python
                    'longest_row': max((x for x in enumerate(nim.rows)), key=lambda
                    ↪  y: y[1])[0],
                    # only 1-stick row and not all rows having only 1 stick
                    '1_stick_row': any([1 for x in nim.rows if x == 1]) and not
                    ↪  all([1 for x in nim.rows if x == 1]),
                    'nim_sum': self.nim_sum(nim)
            }

            brute_force = []
            for move in stats['possible_moves']:
                tmp = deepcopy(nim)
                tmp.nimming_remove(*move)
                brute_force.append((move, self.nim_sum(tmp)))
            stats['brute_force'] = brute_force

            return stats

        def strategy(self):
            '''
            Returns the best move to make based on the statistics
            '''
            def engine(nim: Nim):
                stats = self.statistics(nim)
                if stats['num_active_rows'] == 1:
                    # logging.info('m1')
                    return Nimply(stats['shortest_row'], random.randint(1,
                    ↪  stats['possible_moves'][0][1]))
                elif stats["num_active_rows"] % 2 == 0:
                    # logging.info('m2')
                    if max(nim.rows) == 1:
                        return Nimply(stats['longest_row'], 1)
                    else:
                        pile = random.choice([i for i, x in enumerate(nim.rows)
                        ↪  if x > 1])
                        return Nimply(pile, nim.rows[pile] - 1)
                elif stats['num_active_rows'] == 3:
                    # logging.info('m3')
                    unique_elements = set(nim.rows)
                    # check if 2 rows have the same number of sticks
                    two_rows_with_same_elements = False
                    for element in unique_elements:
                        if nim.rows.count(element) == 2:
                            two_rows_with_same_elements = True
                            break

                    if len(nim.rows) == 3 and two_rows_with_same_elements:
                        # remove 1 stick from the longest row
                        logging.info(nim.rows)
                        return Nimply(stats['longest_row'], max(max(nim.rows) -
                        ↪  nim.rows[stats['shortest_row']], 1))
```

```python
                        else:
                            # do something random
                            return Nimply(*random.choice(stats['possible_moves']))
                    elif stats['num_active_rows'] >= 4:
                        # logging.info('m4')
                        counter = Counter()
                        for element in nim.rows:
                            counter[element] += 1
                        if len(counter) == 2:
                            if counter.most_common()[0][1] == 1:
                                # remove x sticks from the smallest pile until it is
                                ↪   the same size as the other piles
                                return Nimply(stats['shortest_row'],
                                ↪   max(nim.rows[stats['shortest_row']] -
                                ↪   counter.most_common()[1][0], 1))
                        return random.choice(stats['possible_moves'])
                    else:
                        # logging.info('m5')
                        return random.choice(stats['possible_moves'])
            return engine

    def random_agent(self, nim: Nim):
        '''
        Random agent that takes a random move
        '''
        stats = self.statistics(nim)
        return random.choice(stats['possible_moves'])

    def battle(self, opponent, num_games=1000):
        '''
        Battle this agent against another agent
        '''
        wins = 0
        for _ in range(num_games):
            nim = Nim()
            while not nim.goal():
                nim.nimming_remove(*self.play(nim))
                if sum(nim.rows) == 0:
                    break
                nim.nimming_remove(*opponent.play(nim))
            if sum(nim.rows) == 0:
                wins += 1
        return wins

if __name__ == '__main__':
    rounds = 20
    evolved_agent_wins = 0
    for i in range(rounds):
        nim = Nim(5)
        orig = nim.rows
```

```
160                 fixedrule = FixedRuleNim()
161                 engine = fixedrule.strategy()
162
163                 # play against random
164                 player = 0
165                 while not nim.goal():
166                     if player == 0:
167                         move = engine(nim)
168                         logging.info('move of player 1: ', move)
169                         nim.nimming_remove(*move)
170                         player = 1
171                         logging.info("After Player 1 made move: ", nim.rows)
172                     else:
173                         move = fixedrule.random_agent(nim)
174                         logging.info('move of player 2: ', move)
175                         nim.nimming_remove(*move)
176                         player = 0
177                         logging.info("After Player 2 made move: ", nim.rows)
178                 winner = 1 - player
179                 if winner == 0:
180                     evolved_agent_wins += 1
181         logging.info(f'Fixed rule agent won {evolved_agent_wins} out of {rounds}
        ↪  games')
```

Approach 2: Nim-Sum   Will always win

```
1  from copy import deepcopy
2  from itertools import accumulate
3  from operator import xor
4  import random
5  import logging
6  from lib import Nim
7
8  # 3.1: Agent Using Fixed Rules
9  class ExpertNimSumAgent:
10     '''
11     Play the game of Nim using a fixed rule
12     (always leave nim-sum at the end of turn)
13     '''
14     def __init__(self):
15         self.num_moves = 0
16
17     def nim_sum(self, nim: Nim):
18         '''
19         Returns the nim sum of the current game board
20         by taking an XOR of all the rows.
21         Ideally, agent should try to leave nim sum of 0 at the end of turn
22         '''
23         *_, result = accumulate(nim.rows, xor)
```

```
24          return result
25          # return sum([i^r for i, r in enumerate(nim._rows)])
26
27      def play(self, nim: Nim):
28          # remove objects from row to make nim-sum 0
29          nim_sum = self.nim_sum(nim)
30          all_possible_moves = [(r, o) for r, c in enumerate(nim.rows) for o in
            ↪  range(1, c+1)]
31          move_found = False
32          for move in all_possible_moves:
33              replicated_nim = deepcopy(nim)
34              replicated_nim.nimming_remove(*move)
35              if self.nim_sum(replicated_nim) == 0:
36                  nim.nimming_remove(*move)
37                  move_found = True
38                  break
39          # if a valid move not found, return random move
40          if not move_found:
41              move = random.choice(all_possible_moves)
42              nim.nimming_remove(*move)
43
44          # logging.info(f"Move {self.num_moves}: Removed {move[1]} objects from
            ↪  row {move[0]}")
45          self.num_moves += 1
```

4.1.2 Evolved Agent Approach 1

The rules are evolved using a genetic algorithm. A dictionary of strategies is evolved. The key is the rule (scenario/antecedent). The value is the maximum number of sticks to leave on the board in this scenario.

For instance, for rule 1, the value tuned is the in "If one pile, leave a max of x sticks in the pile".

```
rule_strategy = {
    "one_pile": 2,
    "two_piles": 3,
    "three_piles": 3,
    "n_piles": 4
}

# after mutation / crossover
rule_strategy = {
    "one_pile": 3,
    "two_piles": 2,
    "three_piles": 3,
```

Opponent 1	Opponent 2	Win Rate
Evolved	Random	70%

```
    "n_piles": 4
}
```

Mutation essentially swaps the values in the dictionaries. Crossover takes two parents and randomly chooses strategies for different rules. Intuitively, the machine tries to learn the best strategy for each scenario on the board.

```python
1    '''
2    In this file, I will try to implement Nim where there is an evolved set of
     ↪  rules/strategies.
3    For each scenario, I will have a set of rules that will be used to determine the
     ↪  best move.
4    They are obtained from discussion with friends and from the paper "The Game of
     ↪  Nim" by Ryan Julian
5    The rules currently are:
6    1. If one pile, take $x$ number of sticks from the pile.
7    2. If two piles:
8        a. If 1 pile has 1 stick, take x sticks
9        b. If 2 piles have multiple sticks, take x sticks from the larger pile
10   3. If three piles and two piles have the same size, remove all sticks from the
     ↪  smallest pile
11   4. If n piles and n-1 piles have the same size, remove x sticks from the smallest
     ↪  pile until it is the same size as the other piles
12   '''
13
14   from collections import Counter, namedtuple
15   from copy import deepcopy
16   from itertools import accumulate
17   import logging
18   from operator import xor
19   import random
20   from typing import Callable
21
22   from lib import Genome, Nim, Nimply
23
24   class BrilliantEvolvedAgent:
25       def __init__(self):
26           self.num_moves = 0
27           self.OFFSPRING_SIZE = 200
28           self.POPULATION_SIZE = 50
29           self.GENERATIONS = 100
30           self.nim_size = 5
31
32       def nim_sum(self, nim: Nim):
33           '''
34           Returns the nim sum of the current game board
```

```python
                    by taking an XOR of all the rows.
                    Ideally, agent should try to leave nim sum of 0 at the end of turn
                    '''
                    *_, result = accumulate(nim.rows, xor)
                    return result

            def init_population(self, population_size, nim: Nim):
                    '''
                    Initialize population of genomes,
                    key is rule, value is number of sticks to take
                    The rules currently are:
                    1. If one pile, take $x$ number of sticks from the pile.
                    2. If two piles:
                        a. If 1 pile has 1 stick, wipe out the pile
                        b. If 2 piles have multiple sticks, take x sticks from any pile
                    3. If three piles and two piles have the same size, remove all sticks
from the smallest pile
                    4. If n piles and n-1 piles have the same size, remove x sticks from the
smallest pile until it is the same size as the other piles
                    5. If none of the above rules apply, just pick a random pile and take a
random number of sticks
                    '''
                    population = []
                    for i in range(population_size):
                        # rules 3 and 4 are fixed (apply for 3 or more piles)
                        # different strategies for different rules (situations on the board)
                        individual = {
                            'rule_1': [0, random.randint(0, (self.nim_size - 1) * 2)],
                            'rule_2a': [random.randint(0, 1), random.randint(0,
                                (self.nim_size - 1) * 2)],
                            'rule_2b': [random.randint(0, 1), random.randint(0,
                                (self.nim_size - 1) * 2)],
                            'rule_3': [nim.rows.index(min(nim.rows)), min(nim.rows)],
                            'rule_4': [nim.rows.index(max(nim.rows)), max(nim.rows) -
                                min(nim.rows)]
                        }
                        genome = Genome(individual)
                        population.append(genome)
                    return population

            def crossover(self, parent1, parent2, crossover_rate):
                    '''
                    Crossover function to combine two parents into a child
                    '''
                    child = {}
                    for rule in parent1.rules:
                        if random.random() < crossover_rate:
                            child[rule] = parent1.rules[rule]
                        else:
                            child[rule] = parent2.rules[rule]
```

```python
79              return Genome(child)

80

81      def tournament_selection(self, population, tournament_size):
82          '''
83          Tournament selection to select the best genomes
84          '''
85          tournament = random.sample(population, tournament_size)
86          tournament.sort(key=lambda x: x.fitness, reverse=True)
87          return tournament[0]

88

89      def mutate(self, genome: Genome, mutation_rate=0.5):
90          '''
91          Mutate the genome by switching one of the rules (can end up in something
   ↪  stupid like removing more sticks than there are, but this is checked in the
   ↪  strategy function)
92          '''
93          rule = random.choice(list(genome.rules.keys()))
94          # swap some keys
95          if rule == 'rule_1':
96              genome.rules[rule] = [0, random.randint(0, (self.nim_size - 1) * 2)]
97          elif rule == 'rule_2a':
98              genome.rules[rule] = [random.randint(0, 1), random.randint(0,
                 ↪  (self.nim_size - 1) * 2)]
99          elif rule == 'rule_2b':
100             genome.rules[rule] = [random.randint(0, 1), random.randint(0,
                 ↪  (self.nim_size - 1) * 2)]
101         elif rule == 'rule_3':
102             genome.rules[rule] = [random.randint(0, self.nim_size - 1),
                 ↪  random.randint(0, (self.nim_size - 1) * 2)]
103         elif rule == 'rule_4':
104             genome.rules[rule] = [random.randint(0, self.nim_size - 1),
                 ↪  random.randint(0, (self.nim_size - 1) * 2)]
105         return genome
106         # rule = random.choice(list(genome.rules.keys()))
107         # if random.random() < mutation_rate:
108         #     genome.rules[rule] = [random.randint(0, 1), random.randint(0,
                 ↪  self.nim_size * 2)]
109         # return genome
110         # rule = random.choice(list(genome.keys()))
111         # genome[rule] = random.randint(1, 10)

112

113     def statistics(self, nim: Nim):
114         '''
115         Similar to Squillero's cooked function to get possible moves
116         and statistics on Nim board
117         '''
118         stats = {
119             'possible_moves': [(r, o) for r, c in enumerate(nim.rows) for o in
                 ↪  range(1, c + 1) if nim.k is None or o <= nim.k],
```

```python
            # 'possible_moves': [(row, num_objects) for row in
            ↪  range(nim.num_rows) for num_objects in range(1,
            ↪  nim.rows[row]+1)],
            'num_active_rows': sum(o > 0 for o in nim.rows),
            'shortest_row': min((x for x in enumerate(nim.rows) if x[1] > 0),
            ↪  key=lambda y: y[1])[0],
            'longest_row': max((x for x in enumerate(nim.rows)), key=lambda y:
            ↪  y[1])[0],
            # only 1-stick row and not all rows having only 1 stick
            '1_stick_row': any([1 for x in nim.rows if x == 1]) and not all([1
            ↪  for x in nim.rows if x == 1]),
            'nim_sum': self.nim_sum(nim)
        }

        brute_force = []
        for move in stats['possible_moves']:
            tmp = deepcopy(nim)
            tmp.nimming_remove(*move)
            brute_force.append((move, self.nim_sum(tmp)))
        stats['brute_force'] = brute_force

        return stats

    def strategy(self, genome: dict):
        '''
        Returns the best move to make based on the statistics
        '''
        def evolution(nim: Nim):
            stats = self.statistics(nim)
            if stats['num_active_rows'] == 1:
                num_to_leave = genome.rules['rule_1'][1]
                # see which move will leave the most sticks
                most_destructive_move = max(stats['possible_moves'], key=lambda
                ↪  x: x[1])
                if num_to_leave >= most_destructive_move[1]:
                    # remove only 1 stick
                    return Nimply(most_destructive_move[0], 1)
                else:
                    # make the move that leaves the desired number of sticks
                    move = [(row, num_objects) for row, num_objects in
                    ↪  stats['possible_moves'] if nim.rows[row] - num_objects ==
                    ↪  num_to_leave]
                    if len(move) > 0:
                        return Nimply(*move[0])
                    else:
                        # make random move
                        return Nimply(*random.choice(stats['possible_moves']))

            elif stats['num_active_rows'] == 2:
                # rule 2a
```

```python
                     if stats['1_stick_row']:
                         # if there is a 1-stick row, have to choose between wiping it
                         ↪  out or taking from the other row
                         if genome.rules['rule_2a'][0] == 0:
                             # wipe out the 1-stick row
                             logging.info('wiping out 1-stick row')
                             pile = [row for row in range(nim.num_rows) if
                             ↪  nim.rows[row] == 1][0]
                             return Nimply(pile, 1)
                         else:
                             # take out the desired number of sticks from the other
                             ↪  row
                             pile = random.choice([index for index, x in
                             ↪  enumerate(nim.rows) if x > 1])
                             num_objects_to_remove = max(1, nim.rows[pile] -
                             ↪  genome.rules['rule_2a'][1])
                             # move = [(row, num_objects) for row, num_objects in
                             ↪  stats['possible_moves'] if nim.rows[row] -
                             ↪  num_objects == genome.rules['rule_2a'][1]]
                             return Nimply(pile, num_objects_to_remove)
                     # rule 2b
                     # both piles have many elements, take from either the smallest or
                     ↪  the largest pile
                     else:
                         if genome.rules['rule_2b'][0] == 0:
                             # take from the smallest pile
                             pile = stats['shortest_row']
                             num_objects_to_remove = max(1, nim.rows[pile] -
                             ↪  genome.rules['rule_2b'][1])
                             return Nimply(pile, num_objects_to_remove)
                         else:
                             # take from the largest pile
                             pile = stats['longest_row']
                             num_objects_to_remove = max(1, nim.rows[pile] -
                             ↪  genome.rules['rule_2b'][1])
                             return Nimply(pile, num_objects_to_remove)

             elif stats['num_active_rows'] == 3:
                 unique_elements = set(nim.rows)
                 # check if 2 rows have the same number of sticks
                 two_rows_with_same_elements = False
                 for element in unique_elements:
                     if nim.rows.count(element) == 2:
                         two_rows_with_same_elements = True
                         break

                 if len(nim.rows) == 3 and two_rows_with_same_elements:
                     # remove 1 stick from the longest row
                     return Nimply(stats['longest_row'], max(max(nim.rows) -
                     ↪  nim.rows[stats['shortest_row']], 1))
```

```python
201                    else:
202                        # do something random
203                        return Nimply(*random.choice(stats['possible_moves']))
204
205             counter = Counter()
206             for element in nim.rows:
207                 counter[element] += 1
208             if len(counter) == 2:
209                 if counter.most_common()[0][1] == 1:
210                     # remove x sticks from the smallest pile until it is the same
                        ↪  size as the other piles
211                     return Nimply(stats['shortest_row'],
                        ↪  max(nim.rows[stats['shortest_row']] -
                        ↪  counter.most_common()[1][0], 1))
212                 # else:
213                 #     return random.choice(stats['possible_moves'])
214
215             # for large number of piles, general rule to remove all but 1 stick
                ↪  from a random pile
216             if stats["num_active_rows"] % 2 == 0:
217                 if nim.rows[stats['longest_row']] == 1:
218                     return Nimply(stats['longest_row'], 1)
219                 else:
220                     pile = random.choice([i for i, x in enumerate(nim.rows) if x
                        ↪  > 1])
221                     return Nimply(pile, nim.rows[pile] - 1)
222
223             else:
224                 # this is a fixed rule, does not have random component
225                 # rule from the paper Ryan Julian: The Game of Nim
226                 # If n piles and n-1 piles have the same size, remove x sticks
                    ↪  from the smallest pile until it is the same size as the other
                    ↪  piles
227                 # check if only 1 pile has a different number of sticks
228                 # just make a random move if all else fails
229                 return random.choice(stats['possible_moves'])
230         return evolution
231
232     def random_agent(self, nim: Nim):
233         '''
234         Random agent that takes a random move
235         '''
236         stats = self.statistics(nim)
237         return random.choice(stats['possible_moves'])
238
239     def dumb_agent(self, nim: Nim):
240         '''
241         Agent that takes one element from the longest row
242         '''
243         stats = self.statistics(nim)
```

```python
            return (stats['longest_row'], 1)

    def aggressive_agent(self, nim: Nim):
        '''
        Agent that takes the largest possible move
        '''
        stats = self.statistics(nim)
        if stats['num_active_rows'] % 2 == 0:
            return random.choice(stats['possible_moves'])
        else:
            row = stats['longest_row']
            return (row, nim.rows[row])

        # stats = self.statistics(nim)
        # return max(stats['possible_moves'], key=lambda x: x[1])

    def calculate_fitness(self, genome):
        '''
        Calculate fitness by playing the genome's strategy against a random
        agent
        (cannot use nim sum agent as it is too good)
        '''
        wins = 0
        for i in range(5):
            nim = Nim(5)
            player = 0
            engine = self.strategy(genome)
            while not nim.goal():
                if player == 0:
                    move = engine(nim)
                    nim.nimming_remove(*move)
                    player = 1
                else:
                    nim.nimming_remove(*self.random_agent(nim))
                    player = 0
            winner = 1 - player
            if winner == 0:
                wins += 1
        return wins / 5

    def select_survivors(self, population: list, num_survivors: int):
        '''
        Select the best genomes from the population
        '''
        return sorted(population, key=lambda x: x.fitness,
            reverse=True)[:num_survivors]

    def learn(self, population_size=100, mutation_rate=0.1, crossover_rate=0.7,
        nim: Nim = None):
        initial_population = self.init_population(population_size, nim)
```

```
291         for genome in initial_population:
292             genome.fitness = self.calculate_fitness(genome)
293         for i in range(self.GENERATIONS):
294             # logging.info(f'Generation {i}')
295             new_offspring = []
296             for j in range(self.OFFSPRING_SIZE):
297                 parent1 = random.choice(initial_population)
298                 parent2 = random.choice(initial_population)
299                 child = self.crossover(parent1, parent2, crossover_rate)
300                 child = self.mutate(child)
301                 new_offspring.append(child)
302             initial_population += new_offspring
303             initial_population = self.select_survivors(initial_population,
                 ↪ population_size)
304         best_strategy = initial_population[0]
305         return best_strategy
306
307     def battle(self, opponent, num_games=1000):
308         '''
309         Battle this agent against another agent
310         '''
311         wins = 0
312         for _ in range(num_games):
313             nim = Nim()
314             while not nim.goal():
315                 nim.nimming_remove(*self.play(nim))
316                 if sum(nim.rows) == 0:
317                     break
318                 nim.nimming_remove(*opponent.play(nim))
319             if sum(nim.rows) == 0:
320                 wins += 1
321         return wins
322
323 if __name__ == '__main__':
324     rounds = 20
325     evolved_agent_wins = 0
326     for i in range(rounds):
327         nim = Nim(5)
328         orig = nim.rows
329         brilliantagent = BrilliantEvolvedAgent()
330         best_strategy = brilliantagent.learn(nim=nim)
331         engine = brilliantagent.strategy(best_strategy)
332
333         # play against random
334         player = 0
335         while not nim.goal():
336             if player == 0:
337                 move = engine(nim)
338                 logging.info('move of player 1: ', move)
339                 nim.nimming_remove(*move)
```

```
340              player = 1
341              logging.info("After Player 1 made move: ", nim.rows)
342           else:
343              move = brilliantagent.random_agent(nim)
344              logging.info('move of player 2: ', move)
345              nim.nimming_remove(*move)
346              player = 0
347              logging.info("After Player 2 made move: ", nim.rows)
348        winner = 1 - player
349        if winner == 0:
350           evolved_agent_wins += 1
351    logging.info(f'Evolved agent won {evolved_agent_wins} out of {rounds} games')
```

4.1.3 Evolved Agent Approach 2 (Probability Thresholds)

Strategies were originally chosen based on probability thresholds and a random number. The list of probabilities (thresholds) are evolved using a genetic algorithm. *Intuitively, the machine tries to learn the best probability of choosing each strategy, regardless of the rule.*

```
1    thresholds = [p1, p2, p3]
2    if random.random() < p1:
3        # strategy 1...
4    elif random.random() < p2:
5        # strategy 2...
6    else:
7        # strategy 3...
8
9    class GA:
10       ...
11
12   GA.evolve(thresholds)
```

I discussed this approach with both Prof. Squillero and Calabrese. They both agreed that this was worth exploring. However, upon implementing, I realised that tuning probability thresholds produces poor, near-random performance, *as the system is making decisions without any knowledge of the current situation on the board, or any knowledge of the rules.*

```
1    # 3.2: Agent Using Evolved Rules (Randomly Chooses Between Strategies Based
     ↪  on Probabilities)
2    from itertools import accumulate
3    from operator import xor
4    import random
5    import numpy as np
6
```

```python
from lib import Nim

class EvolvedAgent1:
    '''
    Plays Nim using a set of rules that are evolved
    '''
    def __init__(self):
        self.num_moves = 0

    def nim_sum(self, nim: Nim):
        '''
        Returns the nim sum of the current game board
        by taking an XOR of all the rows.
        Ideally, agent should try to leave nim sum of 0 at the end of turn
        '''
        *_, result = accumulate(nim.rows, xor)
        return result

    def play_nim(self, nim: Nim, prob_list: list):
        '''
        GA can choose between the following strategies:
        1. Randomly pick any row and any number of elements from that row
        2. Pick the shortest row
        3. Pick the longest row
        4. Pick based on the nim-sum of the current game board
        '''
        all_possible_moves = [(r, o) for r, c in enumerate(nim.rows) for o in
        ↪   range(1, c+1)]
        strategies = {
            'nim_sum': random.choice([move for move in all_possible_moves if
            ↪   self.nim_sum(deepcopy(nim).nimming_remove(*move)) == 0]),
            'random': random.choice(all_possible_moves),
            'all_elements_shortest_row': (nim.rows.index(min(nim.rows)),
            ↪   min(nim.rows)),
            '1_element_shortest_row': (nim.rows.index(min(nim.rows)), 1),
            'random_element_shortest_row': (nim.rows.index(min(nim.rows)),
            ↪   random.randint(1, min(nim.rows))),
            'all_elements_longest_row': (nim.rows.index(max(nim.rows)),
            ↪   max(nim.rows)),
            '1_element_longest_row': (nim.rows.index(max(nim.rows)), 1),
            'random_element_longest_row': (nim.rows.index(max(nim.rows)),
            ↪   random.randint(1, max(nim.rows))),
        }

        p = random.random()
        strategy = None
        if p < prob_list[0]:
            strategy = strategies['random']
        elif p >= prob_list[0] and p < prob_list[1]:
```

```python
                    strategy =
 ↪ random.choice([strategies['all_elements_shortest_row'],
 ↪ strategies['1_element_shortest_row'],
 ↪ strategies['random_element_shortest_row']])
            elif p >= prob_list[1] and p < prob_list[2]:
                strategy = random.choice([strategies['all_elements_longest_row'],
 ↪ strategies['1_element_longest_row'],
 ↪ strategies['random_element_longest_row']])
            else:
                strategy = strategies['nim_sum']

            nim.nimming_remove(*strategy)
            self.num_moves += 1
            return sum(nim.rows)

    def play(self, nim: Nim):
        '''
        Play the game of Nim using the evolved rules
        '''
        prob_list = [0.25, 0.5, 0.75, 1]
        prob_list = self.evolve_probabilities(nim, prob_list, 20, 5)
        self.play_nim(nim, prob_list)

    def crossover(self, p1, p2):
        '''
        Crossover between two parents
        '''
        return np.random.choice(p1 + p2, size=4, replace=True)

    def evolve_probabilities(self, nim: Nim, prob_list: list,
 ↪ num_generations: int, num_children: int):
        '''
        Evolve the probabilities of the strategies
        '''
        # create initial population
        population = [prob_list for _ in range(num_children)]
        # create initial fitness scores
        fitness_scores = [self.play(nim, p) for p in population]
        # create initial parents
        parents = [population[i] for i in np.argsort(fitness_scores)[:2]]
        # create new population
        new_population = []
        for _ in range(num_generations):
            # create children
            for _ in range(num_children):
                p1 = random.choice(parents)
                p2 = random.choice(parents)
                child = self.crossover(p1, p2)
                # child = []
                # for i in range(len(parents[0])):
```

```
94                      #       # crossover between parents
95
96                      #       child.append(random.choice(parents)[i])
97                    new_population.append(child)
98                # create fitness scores
99                fitness_scores = [self.play_nim(nim, p) for p in new_population]
100               # create new parents
101               parents = [new_population[i] for i in
    ↪  np.argsort(fitness_scores)[:2]]
102               # create new population
103               new_population = []
104           return parents[0]
```

4.1.4 Minmax

In 'minmax.py', the minimax algorithm is implemented. It recursively traverses the game tree to maximise potential returns. As a result, it is a near-optimal strategy that reported '100%' win rate against random opponents.

Since the recursive algorithm is slow:

1. The tree is pruned momentarily, stopping the algorithm from exploring parts of the tree that will not materialise on the game board.

2. A maximum depth is set, so that the recursive loop is stopped when a particular depth is reached.

Although not significant, an '@lru_cache' decorator is applied on the minmax operation after ensuring that the Nim state (row composition) is serializable.

```
1  from copy import deepcopy
2  from functools import lru_cache
3  from itertools import accumulate
4  import math
5  from operator import xor
6  from evolved_nim import BrilliantEvolvedAgent
7  import logging
8  from lib import Nim
9
10 logging.basicConfig(level=logging.INFO)
11
12 class MinMaxAgent:
13     def __init__(self):
14         self.num_moves = 0
15
16     def nim_sum(self, nim: Nim):
17         '''
18         Returns the nim sum of the current game board
19         by taking an XOR of all the rows.
```

```python
20          Ideally, agent should try to leave nim sum of 0 at the end of turn
21          '''
22          *_, result = accumulate(nim.rows, xor)
23          return result
24
25      def evaluate(self, nim: Nim, is_maximizing: bool):
26          '''
27          Returns the evaluation of the current game board
28          '''
29          if all(row == 0 for row in nim.rows):
30              return -1 if is_maximizing else 1
31          else:
32              return -1
33
34      @lru_cache(maxsize=1000)
35      def minmax(self, nim: Nim, depth: int, maximizing_player: bool, alpha: int =
    ↪   -1, beta: int = 1, max_depth: int = 7):
36          '''
37          Depth-limited Minimax algorithm to find the best move with alpha-beta
    ↪ pruning and depth limit
38          '''
39          logging.info("Depth ", depth)
40          if depth == 0 or nim.goal() or depth == max_depth:
41              # logging.info("Depth ", depth)
42              # logging.info("Nim goal ", nim.goal())
43              return self.evaluate(nim, maximizing_player)
44
45          if maximizing_player:
46              value = -math.inf
47              for r, c in enumerate(nim.rows):
48                  for o in range(1, c+1):
49                      # make copy of nim object before running a nimming operation
50                      replicated_nim = deepcopy(nim)
51                      replicated_nim.nimming_remove(r, o)
52                      value = max(value, self.minmax(replicated_nim, depth-1,
    ↪   False, alpha, beta))
53                      alpha = max(alpha, value)
54                      if beta <= alpha:
55                          logging.info("Pruned")
56                          break
57              return value
58          else:
59              value = math.inf
60              for r, c in enumerate(nim.rows):
61                  for o in range(1, c+1):
62                      # make copy of nim object before running a nimming operation
63                      replicated_nim = deepcopy(nim)
64                      replicated_nim.nimming_remove(r, o)
65                      value = min(value, self.minmax(replicated_nim, depth-1, True,
    ↪   alpha, beta))
```

```
66              beta = min(beta, value)
67              if beta <= alpha:
68                  logging.info("Pruned")
69                  break
70          return value
71
72  def play(self, nim: Nim):
73      '''
74      Agent returns the best move based on minimax algorithm
75      '''
76      possible_moves = []
77      for r, c in enumerate(nim.rows):
78          for o in range(1, c+1):
79              # make copy of nim object before running a nimming operation
80              replicated_nim = deepcopy(nim)
81              replicated_nim.nimming_remove(r, o)
82              possible_moves.append((r, o, self.minmax(replicated_nim, 10,
                    ↪ False)))
83      # sort possible moves by the value returned by minimax
84      possible_moves.sort(key=lambda x: x[2], reverse=True)
85      # return the best move
86      return possible_moves[0][0], possible_moves[0][1]
87
88  def battle(self, opponent, num_games=1000):
89      '''
90      Battle this agent against another agent
91      '''
92      wins = 0
93      for _ in range(num_games):
94          nim = Nim()
95          while not nim.goal():
96              nim.nimming_remove(*self.play(nim))
97              if sum(nim.rows) == 0:
98                  break
99              nim.nimming_remove(*opponent.play(nim))
100         if sum(nim.rows) == 0:
101             wins += 1
102     return wins
103
104 if __name__ == "__main__":
105
106     rounds = 10
107
108     minmax_wins = 0
109     for i in range(rounds):
110         nim = Nim(num_rows=5)
111         agent = MinMaxAgent()
112         random_agent = BrilliantEvolvedAgent()
113         player = 0
114         while not nim.goal():
```

```
115              if player == 0:
116                  move = agent.play(nim)
117                  logging.info(f"Minmax move {agent.num_moves}: Removed {move[1]}
     ↪  objects from row {move[0]}")
118                  logging.info(nim.rows)
119                  nim.nimming_remove(*move)
120              else:
121                  move = random_agent.random_agent(nim)
122                  logging.info(f"Random move {random_agent.num_moves}: Removed
     ↪  {move[1]} objects from row {move[0]}")
123                  logging.info(nim.rows)
124                  nim.nimming_remove(*move)
125              player = 1 - player
126
127          winner = 1 - player
128          if winner == 0:
129              minmax_wins += 1
130          # player that made the last move wins
131          logging.info(f"Player {winner} wins in round {i+1}!")
132
133      logging.info(f"Minmax wins {minmax_wins} out of {rounds} rounds")
```

4.1.5 Reinforcement Learning

Both temporal difference learning (TDL) and monte carlo learning (MCL) are
implemented. In TDL, the Q values are updated after each move. In MCL, the
learning is episodic so a goal dictionary is traversed backwards.

State Hashing The state for TDL consists of a key-value dictionary. The rep-
resentation is: (the rows in nim, action tuple): Q. The rows are hashed into a
string, with each value separated by a hyphen. In TDL, Q values are updated
after each move.

Temporal Difference Learning (TDL)

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)$$

TDL exploits the Markov property of the game, where the next state is only
dependent on the current state and the action taken. Performance was initially
poor, but improved after tuning the hyperparameters (alpha, gamma, epsilon).

The best reported win rate is 80% against a random opponent after 5000 rounds
of training at a 0.4 epsilon (exploration rate) and 1000 iterations of testing at 0
epsilon (max exploitation). Learning rate is decayed accordingly.

```python
1  class NimRLTemporalDifferenceAgent:
2      """
3      An agent that learns to play Nim through temporal difference learning.
4      """
5      def __init__(self, num_rows: int, epsilon: float = 0.4, alpha: float = 0.3,
   ↪  gamma: float = 0.9):
6          """Initialize agent."""
7          self.num_rows = num_rows
8          self.epsilon = epsilon
9          self.alpha = alpha
10         self.gamma = gamma
11         self.current_state = None
12         self.previous_state = None
13         self.previous_action = None
14         self.Q = dict()
15
16     def init_reward(self, state: Nim):
17         '''Initialize reward for every state and every action with a random value'''
18         for i in range(1, state.num_rows):
19             nim = Nim(num_rows=i)
20             for r, c in enumerate(nim.rows):
21                 for o in range(1, c+1):
22                     self.set_Q(hash_list(nim.rows), (r, o),
23                                np.random.uniform(0, 0.01))
24
25     def get_Q(self, state: Nim, action: tuple):
26         """Return Q-value for state and action."""
27         if (hash_list(state.rows), action) in self.Q:
28             logging.info("Getting Q for state: {} and action:
   ↪  {}".format(hash_list(state.rows), action))
29             logging.info("Q-value: {}".format(self.Q[(hash_list(state.rows),
   ↪  action)]))
30             return self.Q[(hash_list(state.rows), action)]
31         else:
32             # initialize Q-value for state and action
33             self.set_Q(hash_list(state.rows), action, np.random.uniform(0, 0.01))
34             return self.Q[(hash_list(state.rows), action)]
35
36     def set_Q(self, state: str, action: tuple, value: float):
37         """Set Q-value for state and action."""
38         # logging.info("Setting Q for state: {} and action: {} to value:
   ↪  {}".format(state, action, value))
39         self.Q[(state, action)] = value
40
41     def get_max_Q(self, state: Nim):
42         """Return maximum Q-value for state."""
43         max_Q = -math.inf
44         # logging.info(state.rows)
45         for r, c in enumerate(state.rows):
46             for o in range(1, c+1):
```

```python
47              # logging.info("Just Q: {}".format(self.get_Q(state, (r, o))))
48          max_Q = max(max_Q, self.get_Q(state, (r, o)))
49      # logging.info("Max Q: {}".format(max_Q))
50      return max_Q
51
52  def get_average_Q(self, state: Nim):
53      """Return average Q-value for state."""
54      total_Q = 0
55      for r, c in enumerate(state.rows):
56          for o in range(1, c+1):
57              total_Q += self.get_Q(state, (r, o))
58      return total_Q / len(state.rows)
59
60  def get_possible_actions(self, state: Nim):
61      """Return all possible actions for state."""
62      possible_actions = []
63      for r, c in enumerate(state.rows):
64          for o in range(1, c+1):
65              possible_actions.append((r, o))
66      return possible_actions
67
68  def get_action(self, state: Nim):
69      """Return action based on epsilon-greedy policy."""
70      if random.random() < self.epsilon:
71          return random.choice(self.get_possible_actions(state))
72      else:
73          logging.info("Getting best action")
74          max_Q = -math.inf
75          best_action = None
76          for r, c in enumerate(state.rows):
77              for o in range(1, c+1):
78                  Q = self.get_Q(state, (r, o))
79                  if Q > max_Q:
80                      max_Q = Q
81                      best_action = (r, o)
82          return best_action
83
84  def register_state(self, state: Nim):
85      # for each possible move in state, initialize random Q value
86      for r, c in enumerate(state.rows):
87          for o in range(1, c+1):
88              if (hash_list(state.rows), (r, o)) not in self.Q:
89                  val = np.random.uniform(0, 0.01)
90                  # logging.info("Registering state: {} and action: {} to
                  ↪ {}".format(state.rows, (r, o), val))
91                  self.set_Q(hash_list(state.rows), (r, o), val)
92              else:
93                  logging.info("State already registered: {} and action:
                  ↪ {}".format(state.rows, (r, o)))
94
```

```python
95    def update_Q(self, reward: int, game_over: bool):
96        """Update Q-value for previous state and action."""
97
98        if game_over:
99            # self.set_Q(hash_list(self.previous_state.rows), self.previous_action,
              ↪  reward)
100           self.set_Q(hash_list(self.previous_state.rows), self.previous_action,
              ↪  self.get_Q(self.previous_state, self.previous_action) + self.alpha *
              ↪  (reward - self.get_Q(self.previous_state, self.previous_action)))
101
102       else:
103       # if reward != -1:
104           self.register_state(self.current_state)
105           if self.previous_action is not None:
106               self.set_Q(hash_list(self.previous_state.rows), self.previous_action,
                  ↪  self.get_Q(self.previous_state, self.previous_action) +
107                           self.alpha * (reward + self.gamma) *
                  ↪  (self.get_max_Q(self.current_state) -
                  ↪  self.get_Q(self.previous_state,
                  ↪  self.previous_action)))
108       # else:
109       #     self.set_Q(hash_list(self.previous_state.rows), self.previous_action,
          ↪  self.get_Q(self.previous_state, self.previous_action) + self.alpha *
          ↪  (reward - self.get_Q(self.previous_state, self.previous_action)))
110
111   def print_best_action_for_each_state(self):
112       for state in self.Q:
113           logging.info("State: {}".format(state[0]))
114           nim = Nim(5)
115           nim.rows = unhash_list(state[0])
116           logging.info("Best action: {}".format(self.choose_action(nim)))
117
118   def test_against_random(self, round, random_agent):
119       wins = 0
120       for i in range(rounds):
121           nim = Nim(num_rows=5)
122           player = 0
123           while not nim.goal():
124               if player == 0:
125                   move = self.choose_action(nim)
126                   # logging.info(f"Reinforcement move: Removed {move[1]} objects
                      ↪  from row {move[0]}")
127                   nim.nimming_remove(*move)
128               else:
129                   move = random_agent(nim)
130                   # logging.info(f"Random move {random_agent.num_moves}: Removed
                      ↪  {move[1]} objects from row {move[0]}")
131                   nim.nimming_remove(*move)
132               player = 1 - player
133
```

```
134            winner = 1 - player
135            if winner == 0:
136                wins += 1
137
138        logging.info(f"Win Rate in round {round}: {wins / rounds}")
139
140    def battle(self, agent, rounds=1000, training=True, momentary_testing=False):
141        """Train agent by playing against other agents."""
142        agent_wins = 0
143        winners = []
144        for episode in range(rounds):
145            # logging.info(f"Episode {episode}")
146            nim = Nim(num_rows=5)
147            self.current_state = nim
148            self.previous_state = None
149            self.previous_action = None
150            player = 0
151            while True:
152                reward = 0
153                if player == 0:
154                    self.previous_state = deepcopy(self.current_state)
155                    self.previous_action = self.get_action(self.current_state)
156                    self.current_state.nimming_remove(
157                        *self.previous_action)
158                    player = 1
159                else:
160                    move = agent(self.current_state)
161                    # logging.info("Random agent move: {}".format(move))
162                    self.current_state.nimming_remove(*move)
163                    player = 0
164
165                # learning by calculating reward for the current state
166                if self.current_state.goal():
167                    winner = 1 - player
168                    if winner == 0:
169                        logging.info("Agent won")
170                        agent_wins += 1
171                        reward = 1
172                    else:
173                        logging.info("Random won")
174                        reward = -1
175                    winners.append(winner)
176                    self.update_Q(reward, self.current_state.goal())
177                    break
178                else:
179                    self.update_Q(reward, self.current_state.goal())
180
181            # decay epsilon after each episode
182            self.epsilon = self.epsilon - 0.1 if self.epsilon > 0.1 else 0.1
183            self.alpha *= -0.0005
```

```
184        if self.alpha < 0.1:
185            self.alpha = 0.1
186
187        if training and momentary_testing:
188            if episode % 100 == 0:
189                logging.info(f"Episode {episode} finished, sampling")
190                random_agent = BrilliantEvolvedAgent()
191                self.test_against_random(
192                    episode, random_agent.random_agent)
193
194    if not training:
195        logging.info("Reinforcement agent won {} out of {} games".format(
196            agent_wins, rounds))
197    # self.print_best_action_for_each_state()
198    return winners
199
200 def choose_action(self, state: Nim):
201     """Return action based on greedy policy."""
202     max_Q = -math.inf
203     best_action = None
204     for r, c in enumerate(state.rows):
205         for o in range(1, c+1):
206             Q = self.get_Q(state, (r, o))
207             if Q > max_Q:
208                 max_Q = Q
209                 best_action = (r, o)
210     if best_action is None:
211         return random.choice(self.get_possible_actions(state))
212     else:
213         return best_action
214
215 if __name__ == "__main__":
216 rounds = 10000
217 minmax_wins = 0
218
219 nim = Nim(num_rows=5)
220 agent_tda = NimRLTemporalDifferenceAgent(num_rows=5)
221 random_agent = RandomAgent()
222
223 # agentG = NimRLMonteCarloAgent(num_rows=7)
224 agent_tda.battle(random_agent.play, rounds=10000)
225 agent_tda.epsilon = 0.1
226
227 # TESTING
228 logging.info("Testing against random agent")
229 agent_tda.battle(random_agent.random_agent, training=False, rounds=1000)
```

Monte Carlo Learning

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(G - Q(s, a) \right)$$

In MCL, the learning is episodic so a goal dictionary is traversed backwards. MCL takes a more holistic approach to learning, where rewards are based on every past move.

```python
logging.basicConfig(level=logging.INFO)

def hash_list(l):
    '''
    Hashes a list of integers into a string
    '''
    return "-".join([str(i) for i in l])


def unhash_list(l):
    '''
    Unhashes a string of integers into a list
    '''
    return [int(i) for i in l.split("-")]


def decay(value, decay_rate):
    return value * decay_rate


class NimRLMonteCarloAgent:
    def __init__(self, num_rows: int, epsilon: float = 0.3, alpha: float = 0.5,
      gamma: float = 0.9):
        """Initialize agent."""
        self.num_rows = num_rows
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.current_state = None
        self.previous_state = None
        self.previous_action = None
        self.G = dict()
        self.state_history = []

    def get_action(self, state: Nim):
        """Return action based on epsilon-greedy policy."""
        if random.random() < self.epsilon:
            action = random.choice(self.get_possible_actions(state))
            if (hash_list(state.rows), action) not in self.G:
                self.G[(hash_list(state.rows), action)] = random.uniform(1.0,
                  0.01)
            return action
        else:
            max_G = -math.inf
            best_action = None
```

```python
                for r, c in enumerate(state.rows):
                    for o in range(1, c+1):
                        if (hash_list(state.rows), (r, o)) not in self.G:
                            self.G[(hash_list(state.rows), (r, o))] =
                            ↪  random.uniform(1.0, 0.01)
                            G = self.G[(hash_list(state.rows), (r, o))]
                        else:
                            G = self.G[(hash_list(state.rows), (r, o))]
                        if G > max_G:
                            max_G = G
                            best_action = (r, o)
            return best_action

    def update_state(self, state, reward):
        self.state_history.append((state, reward))

    def learn(self):
        target = 0

        for state, reward in reversed(self.state_history):
            self.G[state] = self.G.get(state, 0) + self.alpha * (target -
            ↪  self.G.get(state, 0))
            target += reward

        self.state_history = []
        self.epsilon -= 10e-5

    def compute_reward(self, state: Nim):
        return 0 if state.goal() else -1

    def get_possible_actions(self, state: Nim):
        actions = []
        for r, c in enumerate(state.rows):
            for o in range(1, c+1):
                actions.append((r, o))
        return actions

    def get_G(self, state: Nim, action: tuple):
        return self.G.get((hash_list(state.rows), action), 0)

    def battle(self, opponent, training=True):
        player = 0
        agent_wins = 0
        for episode in range(rounds):
            self.current_state = Nim(num_rows=self.num_rows)
            while True:
                if player == 0:
                    action = self.get_action(self.current_state)
                    self.current_state.nimming_remove(*action)
                    reward = self.compute_reward(self.current_state)
```

```
 92                      self.update_state(hash_list(self.current_state.rows), reward)
 93                      player = 1
 94                  else:
 95                      action = opponent(self.current_state)
 96                      self.current_state.nimming_remove(*action)
 97                      player = 0
 98
 99                  if self.current_state.goal():
100                      logging.info("Player {} wins!".format(1 - player))
101                      break
102
103              winner = 1 - player
104              if winner == 0:
105                  agent_wins += 1
106              # episodic learning
107              self.learn()
108
109              if episode % 1000 == 0:
110                  logging.info("Win rate: {}".format(agent_wins / (episode + 1)))
111          if not training:
112              logging.info("Win rate: {}".format(agent_wins / rounds))
```

4.2   Acknowledgements

4.3   Received Reviews

> **Xiusss**
> --
> Hi! Your code is really clean. There are a lot of useful and really detailed comments. Monte Carlo method is a good choice, well done! Despite it didn't give you the outcome you expected, I found the approach referred to as "approach 2" of task 3.2 really interesting.
> NIce!

Francesco Sattolo

Design considerations:

- The rule based agent works correctly
- The first evolution approach is very interesting since it evolves taking into consideration the current state of the board.
- The second evolution approach is similar to what I've done so good job coming up with both - In the fitness function maybe you could also make it compete with different strategies and not only with pure_random, so that it can improve more. You could also consider different Nim games with different size, to face a bigger variety of situations - With the minmax agent some strategies can be implemented to improve performances with bigger Nim games (for example considering as equal different Nim games like 1,2,3,4 and 1,2,4,3) - Very good job with the reinforcement learning agent

Implementation considerations:
- Executing the code as it is does not produce any output for me, I managed to see some output by replacing logging.info invocations with print. The reason, for example in fixed_rules_nim.py is that the line logging.basicConfig(level=logging.INFO) is missing, and sometimes you use the "print syntax" for the parameters, which is not accepted by the logging library (('move of player 1: ', move)). My suggestion is to always use f-strings, since they are accepted by both print and logging.info and are very powerful and easy to use.
- There are some "copy-paste" oversights, like the init_population which is not used in the fixed_rule_nim.py or some variable names.
- There is no way to see the ExpertNimSumAgent in action.
- For the ExpertNimSumAgent there is a way to compute the best move (the one that brings the nim sum=0) without bruteforcing it, which will improve performance. You can find it in my repository.
- *_, result = accumulate(state.rows, xor) can be replaced by result = reduce(state.rows, xor)
- In the evaluate function of the MinMaxAgent you could use the goal function that you defined for the Nim class for consistency.
- Hardcoding lru cache size of 1000 would probably not contain many possible states when working with big games.
- You use 7 as max hardcoded depth, but actually you start with depth = 10 and remove 1 depth at every iteration. This effectively means that you only go 3 layers deep, which only allow you to solve very small Nim games.
- Well written readme

4.4 Given Reviews

4.4.1 Karl

Karl's code (irrelevant parts/utility functions removed):

```python
"""
Agents based on different strategies playing Nim (description here:
    https://en.wikipedia.org/wiki/Nim)
    1. Agent based on rules
    2. Agent based on evolved rules
    3. Agent using minmax
    4. Agent using reinforcement learning

@Author: Karl Wennerström in collaboration with Erik Bengtsson (s306792)
"""

# ...

# %% Q.2 Create own strategy based on cooked information

# strategy maker: play by the rules
def make_strategy(agent: Evolvable_agent) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        # rule 1
        if data['active_rows_number'] == 1:
            row, elem = agent.rule1(data)
            ply = Nimply(row, elem)

        elif data['one_multiple_elem_row']:  # all rows but one have a single
            elem

            # rule2
            if data['active_rows_number'] % 2 == 0:  # even rows
                row, elem = agent.rule2(data)
                ply = Nimply(row, elem)

            # rule 3
            else:  # odd rows
                row, elem = agent.rule3(data)
                ply = Nimply(row, elem)

        elif not data['one_multiple_elem_row']:  # multiple rows are with
            multiple elems (or also only ones)

            # rule 4
            if data['active_rows_number'] % 2 == 0:
                row, elem = agent.rule4(data)
```

```python
42                         ply = Nimply(row, elem)

43
44                     # rule 5
45                     else:
46                         row, elem = agent.rule5(data)
47                         ply = Nimply(row, elem)

48
49
50             else:
51                 # rule 6 (will we ever get here?)
52                 logging.info(f'RULE 6!!! Board = {state.rows}')
53                 row, elem = agent.rule6(data)
54                 ply = Nimply(row, elem)

55
56             return ply

57
58     return evolvable

59

60
61 # human strategy, make moves through input
62 def my_strategy(state: Nim) -> Nimply:
63     print(f'Current state: {state.rows}')
64     data = cook_status(state)
65     pm = data['possible_moves']
66     index = input(f'Choose a play: {[(i, m) for i, m in enumerate(pm)]}')
67     while True:
68         try:
69             assert int(index) in range(len(pm))
70         except Exception:
71             print('Invalid input, try again')
72             index = input(f'Choose a play: {[(i, m) for i, m in enumerate(pm)]}')
73         else:
74             row = pm[int(index)][0]
75             elems = pm[int(index)][1]
76             break
77     return Nimply(row, elems)

78

79
80 # dumb strategy (to evaluate my agent)
81 def dumb_agent(state: Nim) -> Nimply:
82     """
83     Make stupid move. Always remove 1 from shortest row
84     """
85     data = cook_status(state)
86     row = data['shortest_row']
87     return Nimply(row, 1)

88

89
90 # random strategy (to evaluate my agent)
91 def pure_random(state: Nim) -> Nimply:
```

```python
92          """Agent playing completely random"""
93          row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
94          num_objects = random.randint(1, state.rows[row])
95          return Nimply(row, num_objects)
96
97
98   def semi_smart(state: Nim) -> Nimply:
99          """ Make use of rule 1-3, else random"""
100         data = cook_status(state)
101
102         if data['active_rows_number'] == 1:
103             row = data['active_rows_index'][0]
104             elems = state.rows[row]
105             ply = Nimply(row, elems)
106
107         elif data['one_multiple_elem_row']:  # all rows but one have a single elem
108             if data['active_rows_number'] % 2 == 0:
109                 move = [(r, e) for (r, e) in data["possible_moves"] if state.rows[r]
                      ↪  - e == 1][0]
110                 ply = Nimply(move[0], move[1])
111             else:
112                 move = [(r, e) for (r, e) in data["possible_moves"] if
113                         state.rows[r] - e == 0 and r not in
                            ↪  data['single_elem_rows_index']][0]
114                 ply = Nimply(move[0], move[1])
115         else:
116             row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
117             num_objects = random.randint(1, state.rows[row])
118             ply = Nimply(row, num_objects)
119         return ply
120  # %% EVOLUTION STRATEGY DESCRIBED
121
122  """
123  (mu, lambda)-strategy
124      1. Create population with the same set of rules but different parameters for
     ↪  each rule
125      2. k individuals competes in a tournament where the winner becomes a parent
126      3. Perform cross_over between two parents and mutate (aggregate random rule,
     ↪  e.g. mean(both parents' rule)) with certain prob
127      4. Generate offspring where OFFSPRING_SIZE>>POPULATION_SIZE
128      5. Fitness for offsprings corresponds to how many games are won against their
     ↪  'siblings'
129      6. Slice new population from fittest offpring
130      7. Repeat step 2-6 GENERATION times
131  """
132
133  # %% Evolution strategy-functions
134  def init_population():
135      """Initialize population"""
136      pop = []
```

```python
137         for i in range(POPULATION_SIZE):
138             pop.append(Evolvable_agent(NIM_SIZE))
139         return pop
140
141
142     def calc_fitness(individuals: list) -> None:
143         """Calculate fitness for each individual as a proportion of won games against
            ↪  different opponents"""
144         for ind in individuals:
145             fitness = []
146             for idx, strat in enumerate(OPPONENTS):
147                 wins = 0
148                 for match in range(NUM_MATCHES):
149                     wins += head2head(ind, strat)
150                 fitness.append(wins / NUM_MATCHES)
151             ind.fitness = tuple(fitness)
152
153
154     # compute fitness by head2head-games
155     def head2head(agent: Evolvable_agent, opponent: Callable):
156         """One game between evolvable agent and opponent"""
157         players = (make_strategy(agent), opponent)
158
159         nim = Nim(NIM_SIZE)
160         player = 0
161         while nim:
162             ply = players[player](nim)
163             nim.nimming(ply)
164             player = 1 - player
165         winner = 1 - player
166         if winner == 0:
167             return 1
168         else:
169             return 0
170
171     def fittest_individuals(pop: list) -> list:
172         """Return the most fit individuals to use in offspring generation"""
173         return sorted(pop, key=lambda l: l.fitness, reverse=True)[:POPULATION_SIZE]
174
175
176     # tournament to decide parents
177     def tournament(population: list, k: int) -> dict:
178         """Select best individual out of k competing in a tournament"""
179         contestors = random.sample(population, k=k)
180         best_contestor = sorted(contestors, key=lambda l: l.fitness, reverse=True)[0]
181         return best_contestor
182
183
184     def cross_over(parent1: Evolvable_agent, parent2: Evolvable_agent, mutation_prob:
        ↪  float) -> Evolvable_agent:
```

```python
        """Generate new individual by cross-over of parents' rules"""
        rules = [rule for rule in parent1.rules.keys()]
        new_rules = {}
        child = Evolvable_agent(NIM_SIZE)
        for k in rules:
            which_parent = random.randint(1, 2)
            new_rules[k] = parent1.rules[k] if which_parent == 1 else
             ↪   parent2.rules[k]
        if random.random() < mutation_prob:
            rule = random.choice(rules)
            if rule == 'rule_1':
                new_rules[rule] = random.randint(0, (NIM_SIZE - 1) * 2)
            else:
                new_rules[rule] = [random.randint(0, 1), random.randint(0, (NIM_SIZE
                 ↪   - 1) * 2)]
        child.rules = new_rules
        return child


def create_offspring(population: list, k: int, mutation_prob: float) -> list:
    """Create new offspring"""
    offspring = []
    for _ in range(OFFSPRING_SIZE):
        p1 = tournament(population=population, k=k)
        p2 = tournament(population=population, k=k)
        child = cross_over(parent1=p1, parent2=p2, mutation_prob=mutation_prob)
        offspring.append(child)
    return offspring


def get_next_generation(offspring: list) -> list:
    """Find the best individuals in the new generation"""
    calc_fitness(offspring)
    return fittest_individuals(offspring)


# %% PLAYING FUNCTIONS
def evaluate(strategy1: Callable, strategy2: Callable) -> float:
    """Play two strategies against each other and evaluate their performance """
    players = (strategy1, strategy2)
    won = 0

    for m in range(EVAL_MATCHES):
        nim = Nim(NIM_SIZE)
        player = 0
        while nim:
            ply = players[player](nim)
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
```

```
233                won += 1
234        print(f'{strategy1.__name__} wins {won*100/EVAL_MATCHES} % of the games
           ↪  against {strategy2.__name__}')
235        return won / EVAL_MATCHES
236
237
238  def play_nim(strategy1, strategy2):
239        """A visualized match between two strategies"""
240        strategy = (strategy1, strategy2)
241        nim = Nim(NIM_SIZE)
242        logging.debug(f"status: Initial board  -> {nim}")
243        player = 0
244        while nim:
245            ply = strategy[player](nim)
246            nim.nimming(ply)
247            logging.debug(f"status: After player {player} -> {nim}")
248            player = 1 - player
249        winner = 1 - player
250        logging.info(f"status: Player {winner} won!")
251  # %% Q3 - MINMAX AGENT
252
253  """
254      Build a minmax agent that alwasy minimizes the opponents maximum win
255      Play against optimal strategy, should be able to win if start
256      Build as class or function?
257      Need:
258          keep value for each state (exhaustive)
259          condition: return 1 if win -1 else
260          condition: return 0 if not decided
261              play intil determined and traverse back to that state
262  """
263  # %% MINMAX fcn
264  def minmax(state: Nim, my_turn: bool, alpha=-1, beta=1):
265      if not state:  # empty board then I lose
266          return -1 if my_turn else 1
267
268      data = cook_status(state)
269      possible_new_states = []
270      for ply in data['possible_moves']:
271          tmp_state = deepcopy(state)
272          tmp_state.nimming(ply)
273          possible_new_states.append(tmp_state)
274      if my_turn:
275          bestVal = -np.inf
276          for new_state in possible_new_states:
277              value = minmax(new_state, False, alpha, beta)
278              bestVal = max(bestVal, value)
279              alpha = max(alpha, bestVal)
280              if beta <= alpha:
281                  logging.info(f'Pruned')
```

```python
                break
        return bestVal
    else:
        bestVal = np.inf
        new_state = deepcopy(state)
        ply = optimal_strategy(new_state)
        new_state.nimming(ply)
        value = minmax(new_state, True, alpha, beta)
        bestVal = min(bestVal, value)
        return bestVal

def best_move(state: Nim):
    data = cook_status(state)
    for ply in data['possible_moves']:
        tmp_state = deepcopy(state)
        tmp_state.nimming(ply)
        score = minmax(tmp_state, my_turn=False)
        if score > 0:
            break
    return ply

# %% Q4 - RL

"""
Reinforcement learning agent to play Nim

Idea:
    Play using Upper Confidence Trees (UCT), a Monte Carlo Tree Search (MCTS)
    algorithm, popular when trade-off between
    finding best-so-far and finding a better one

Need:
    * All possible states (TODO: sort state so that e.g. 1 1 0 == 1 0 1)
        * Init with value 0 and visits 0
    * Actions for each state (based on data)
    * Simulate function
    * Reward function

Outline:
    1. Selection (select an unvisited node) with highest UCT
    2. Expand to that node
    3. Simulate from that node until termination
    4. Backpropagate and update node with statistics
        * N(v) - number of visits for node v
        * Q(v) - value/reward playing from that node

UCT:
    uct(v_i, v) = Q(v_i)/N(v_i) + c*sqrt(log(N(v))/N(v_i)), which prefers child
    nodes with small N(v_i)
```

```python
329        choose action according to highest uct value (init with np.inf to explore
    ↪   every move)
330    """
331
332    # Imports
333    import itertools
334
335
336    # Class
337
338    class RLAgent:
339
340        # INITIALIZATION
            ↪   ----------------------------------------------------------------
341        def __init__(self, nim_size: int, random_factor=0.2,
342                     exploration_factor=np.sqrt(2)):   # explore with 20%, exploit
                        ↪   with 80%
343            self.nim_size = nim_size
344            self.current_state = None
345            self.previous_state = None
346            self.__init_states(nim_size)
347            self.random_factor = random_factor
348            self.c = exploration_factor
349
350        def __init_states(self, nim_size: int):
351            """find all possible board positions"""
352            states = {}
353            rows = [i * 2 + 1 for i in range(nim_size)]
354            elem_ranges = list(itertools.combinations([range(n + 1) for n in rows],
                ↪   r=nim_size))
355            all_states = list(itertools.product(*elem_ranges[0]))
356
357            for state in all_states:
358                states[state] = {}
359                states[state]['visits'] = 0
360                states[state]['value'] = 0
361                states[state]['child_states'] = self.__init_child_states(state)
362            self.states = states
363            # last state is the initial board
364            self.current_state = all_states[-1]
365            self.states[self.current_state]['visits'] = 1
366
367        def __init_child_states(self, state):
368            """Find all states accessible from state"""
369            nim = Nim(self.nim_size)
370            nim._rows = list(state)
371            if nim:
372                data = cook_status(nim)
373                children = []
374                for ply in data['possible_moves']:
```

```
375             tmp_nim = deepcopy(nim)
376             tmp_nim.nimming(ply)
377             children.append(tmp_nim.rows)
378         return children
379
380     # MCTS ----------------------------------------------------------------
381     def selection(self):
382         """Select next move according to highest uct score"""
383         next_state = self.__state_with_highest_uct()
384         return next_state
385
386     def __state_with_highest_uct(self):
387         """Move to child node with highest UCT score (depending on parent and
        ↪   child nodes) """
388         visits_parent = self.states[self.current_state]['visits']
389         best_state = None
390         best_uct = -np.inf
391         for child_state in self.states[self.current_state]['child_states']:
392             visits_child = self.states[child_state]['visits']
393             wins_child = self.states[child_state]['value']
394             uct = wins_child / (visits_child + 1) + self.c *
            ↪   (np.log(visits_parent) / (visits_child + 1)) ** (1 / 2)
395             if uct > best_uct:
396                 best_uct = uct
397                 best_state = child_state
398         return best_state
399
400     def random_selection(self):
401         """Explore and move to random state"""
402         next_state =
        ↪   random.choice(tuple(self.states[self.current_state]['child_states']))
403         return next_state
404
405     def expand(self, next_state):
406         """Expand to the found next state. Return the ply that takes agent
        ↪   there"""
407         self.previous_state = self.current_state
408         self.current_state = next_state
409         ply = self.__next_ply()
410         return ply
411
412     def __next_ply(self):
413         """ Find ply that takes agent from previous state to current state"""
414         # manipulate nim
415         nim = Nim(self.nim_size)
416         nim._rows = list(self.previous_state)
417         data = cook_status(nim)
418         ply = [ply for ply in data['possible_moves'] if data['rows'][ply[0]] -
        ↪   ply[1] == self.current_state[ply[0]]][0]
419         return ply
```

```python
420
421    def simulate(self, opponent: Callable, n_matches: int):
422        """Simulate game of nim vs opponent by letting RL agent play randomly
           ↪  from current state"""
423        players = (opponent, pure_random)  # rl agent is second since played move
           ↪  to get here
424        nim = Nim(self.nim_size)
425        won = 0
426        for match in range(n_matches):
427            # forbidden stuff
428            nim._rows = list(self.current_state)  # play from current state
429
430            player = 0
431            while nim:
432                ply = players[player](nim)
433                nim.nimming(ply)
434                player = 1 - player
435            if player == 0:
436                won += 1
437
438        # update results
439        self.backpropagate(n_matches, won)
440
441    def backpropagate(self, visits: int, reward: int):
442        """Update results after simulating `visits` times game from current
           ↪  state"""
443        self.states[self.current_state]['visits'] += visits
444        self.states[self.current_state]['value'] += reward
445
446    # TRAINING ----------------------------------------------------------------
447    def learn_to_play(self, opponents: list, n_sims: int, n_matches: int):
448        """Simulate the game from original state. For each move, simulate the
           ↪  outcome n_matches times.
449        Keep moving until board is empty, then repeat n_sims times."""
450        for opponent in opponents:
451            for n in tqdm(range(n_sims), desc="Iterations, %s"
               ↪  %opponent.__name__):
452                # always start from initial state in a new simulation
453                nim = Nim(self.nim_size)
454                self.current_state = nim.rows
455
456                while nim:
457                    if random.random() < self.random_factor:
458                        # choose random state
459                        ns = self.random_selection()
460                    else:
461                        ns = self.selection()
462                    ply = self.expand(next_state=ns)
463                    nim.nimming(ply)
464
```

```python
465                        self.simulate(opponent, n_matches)
466
467        def get_statistics(self):
468            """Print overview of number of visits and wins for a visited state"""
469            info = [(k, v['value'], v['visits']) for k, v in self.states.items()]
470            for state in info:
471                if state[2] > 0:   # at least 1 visit
472                    print(f'State {state[0]}: \tvisits {state[2]} \twins {state[1]}')
473
474        def policy(self, state: Nim) -> Nimply:
475            """The policy, i.e. the next move for the current state"""
476            self.current_state = state.rows
477            ns = self.selection()
478            ply = self.expand(next_state=ns)
479            return ply
480
481    # %% MAIN
482    import argparse
483
484    if __name__ == '__main__':
485
486        # VARIABLES
487        NIM_SIZE = 3
488        NUM_MATCHES = 100
489        EVAL_MATCHES = 100
490
491        # INPUT
492        parser = argparse.ArgumentParser()
493        parser.add_argument("-t", "--task", dest="task", default=1,
494                            help="Which task should run? Choose from 1, 2, 3 or 4.",
                            ↪   type=int)
495
496        args = parser.parse_args()
497        print(f"Task: {args.task}")
498
499        # -------------------------------TASK 1 - PLAYING THE OPTIMAL STRATEGY
           ↪   ----------------------------
500        if args.task == 1:
501            play_nim(optimal_strategy, optimal_strategy)
502            # play the nim-sum strategy
503            starting_wins = evaluate(optimal_strategy, optimal_strategy)
504            print(f'Optimal strategy wins {starting_wins * 100: .0f}% when starting
               ↪   and {(1 - starting_wins) * 100: .0f}% when not starting.')
505
506        # -------------------------------TASK 2 - EVOLVE AN AGENT
           ↪   ----------------------------
507        elif args.task == 2:
508            # set params
509            POPULATION_SIZE = 50
510            OFFSPRING_SIZE = 200
```

```
511         GENERATIONS = 10
512         OPPONENTS = [dumb_agent, pure_random, semi_smart, optimal_strategy]
513
514         tournament_size = 10
515         mutation_prob = 0.3
516
517         pop = init_population()
518
519         for gen in tqdm(range(GENERATIONS), desc='Generations'):
520             calc_fitness(pop)
521             offspring = create_offspring(pop, tournament_size, mutation_prob)
522             pop = get_next_generation(offspring)
523
524     # ------------------------- TASK 3 - MINMAX FUNCTION
    ↪   -------------------------
525     elif args.task == 3:
526         import time
527         start = time.time()
528         play_nim(best_move, optimal_strategy)
529         elapsed = time.time() - start
530         print(f'It take {elapsed :.2f} seconds to play a game of Nim with size
    ↪   {NIM_SIZE}')
531
532     # ------------------------- TASK 4 - REINFORCEMENT LEARNING
    ↪   -------------------------
533     elif args.task == 4:
534         ITERS = 1000
535
536         # must have run with -t 2 to have a pop
537         if 'pop' in locals():
538             opponents = [pure_random, semi_smart, make_strategy(pop[0]),
    ↪   optimal_strategy]
539         else:
540             opponents = [pure_random, semi_smart, optimal_strategy]
541
542         for opponent in opponents:
543             rl_agent = RLAgent(NIM_SIZE)
544             rl_agent.learn_to_play([opponent], n_sims=ITERS,
    ↪   n_matches=NUM_MATCHES)
545             evaluate(rl_agent.policy, opponent)
546
547
548     else:
549         print(f'Have not finished task {args.task}')
```

Hi Karl,

Here's my review of your lab 3. I have nothing to say about the nim-sum agent, so I'll focus on the rest.

1. There is a single file with the solutions for all labs. To improve readability,

consider modularising by having a shared library file and a class for each task.

2. I like that you have the option to play your agents against a human. I wish I also did this, as it's interesting to run.

3. The README is very well written and the code is well documented with comments in the right places. I had no issues understand your rules for the evolvable agent, especially since the rules were both explained and linked to individual lines of code.

Evolutionary Algorithm

1. The rules are neat in the sense that rules 4, 5 and 6 are very generalised and will apply to any setup on the board that does not match rules 1, 2 and 3. Hence, the agent always has something to fall back on, without resorting to a completely random move. However, the rules you implemented are a small subset of a much larger collection in the literature. A few extra rules can be added to cater to very specific scenarios like "one row left with 2 elements", or a compound rule like: "if one row has x elements" and "another row has 1 elements", then "remove 1 element from the last row". I understand that there are an infinite number of possibilities, but hardcoding a few more for a small nim size is harmless.

2. I like that you modularised your agent with different methods for each rule. It really cleans up the 'if-else' series code block. This is something I didn't do and I will take inspiration from keeping the agent as a separate class.

3. Your mutation strategy to average two genome dictionary values instead of simply swapping them is interesting and may result in fewer cases where the mutated value is unusually small/large for a particular rule. I'll definitely take inspiration from this.

Minimax

1. Your minimax implementation is quite standard and works to near-optimal performance. Apart from alpha-beta pruning, you could also consider limiting the depth to speed up computation for large nim sizes.

Reinforcement Learning

1. I just learnt about Upper Confidence Trees after reading your code, where it seems to resemble some form of tree search. The best children are identified with RL by running the game from that particular state during learning. All in all, this is very well implemented.

2. My only suggestion is to decay/adjust the random_factor during each match. I found that adjusting the exploration epsilon rendered better performance when decayed, favouring exploration at the start and exploitation towards the end. This is just an idea, am not sure how it will work for UCTs.

Overall, good job!

Best, Sidharrth

4.4.2   Jaco

Hi Jaco, here's my review of your lab 3. I watched your presentation in class. Notable points:

1. The README is well-explained, I didn't have much of a problem understanding which strategies were better than others.

2. I also used temporal difference learning as my RLAgent agent for the last task, and I think it is a suitable implementation in this case, as there are not many possible Nim states to consider.

Things to look at:

1. For the GA, I notice that you use a mutation rate of '0.5' that stays constant throughout training. You could consider decaying the value, as I, along with others in the Telegram group, found that high mutation rates at the start were detrimental to training.

2. The computational cost of min-max pruning is vast, so maybe you could consider implementing alpha beta pruning to speed up the process.

3. While your RL agent's implementation is sound, I wonder why your win rate against random is only 48%. You could run for more iterations to see if the win rate improves.

5 Final Project

The purpose of the final project is to implement an efficient agent that can play and win Quarto. Quarto is a multi-player game where 2 players take turns placing pieces on a 4x4 board. The first player to place a piece that satisfies a winning condition wins the game. In my version of Quarto, I consider it to be a two-player game where my agent plays against a random opponent.

5.1 Acknowledgements

Throughout this project, I have discussed with Diego Gasco (s296762). We started the project by discussing ideas for strategies.

- After I tried to get a working Deep Q-Network and realised that it wasn't converging in reasonable time, we discussed the possibility of using some tree search algorithm. It was Diego who suggested MCTS.

- When realising that MCTS can be quite slow towards the end of the game, I suggested building a hybrid QL-MCTS player that would use a base Q-table to remember the best moves so the quadratically complex tree wouldn't need to store so many nodes.

- We later built a hardcoded agent using different rules and realised that it performed very well, and was quick to make a move.

- We then decided to combine everything we did into a hybrid agent that would switch between strategies depending on the board score.

- Diego suggested a good scoring function. He also suggested that a genetic algorithm could be used for this.

- I suggested finding score thresholds to switch between strategies using a genetic algorithm.

While we follow the same hybrid strategy, our code is quite different, apart from a few shared utility functions.

The code for this project is available on Github.

5.2 My Strategy For Solving the Problem

5.2.1 Step 1: Implement and Tune Multiple Search Algorithms

The following algorithms are implemented and the **best performing ones are combined to create a final, hybrid agent that balances speed and efficiency**:

- **Random**: This agent randomly selects positions and pieces on the board. In the spirit of true randomness, it does not take into account the current state of the board.

- **Parameterized Hardcoded Play**: This agent has a set of fixed rules, where it attempts to build a line of like pieces. Where it cannot, it attempts

- **Deep Q-Learning**: This agent uses a deep neural network to approximate the Q-function. It uses a replay buffer to store the experience tuples and uses a target network to stabilize the training process. The agent uses an epsilon-greedy policy to balance exploration and exploitation. The input to the linear network is a flatten list of 1x16 pieces based on the current board composition, while the input to the convolutional neural network is a 4x4x4 board composition.

- **Q-Learning (Temporal Difference Learning)**: This agent uses a Q-table to store the Q-values. It uses a replay buffer to store the experience tuples and uses a target network to stabilize the training process. The agent uses an epsilon-greedy policy to balance exploration and exploitation. A custom OpenAI Gym environment is created to make training easier.

- **Monte Carlo Tree Search**: This agent uses a Monte Carlo Tree Search algorithm to select the best move. It uses a UCB1 formula to select the best child node at each iteration.

- **QL-MCTS**: This algorithm uses a Q-table as it's base and uses a rolled out Monte Carlo Search Tree for a more efficient search. When a state cannot be found in the Q-table, the agent once again goes to the Monte Carlo Tree Search algorithm to find the best move.

The following algorithms failed, producing only a near-random win rate after several hours of training:

1. **Pure Q-Learning**: This agent stores moves made in a Q-table and could not perform feasibly in a test environment even after hours of training, growing it's Q-table and implementing board symmetries.

2. **Deep Q-Learning (Linear and Convolutional Neural Network)**: In this approach, I train a 4-layer deep neural network to predict the Q-values of a given state. Despite several hours of training and hyperparameter tuning (changing the number of layers, optimiser, learning rate), the agent could only reach a 60% win rate in its best attempt. I also tried a convolutional

neural network to feed the entire board composition as a 4x4x4 input (third dimension is the piece attribute), but training was far too slow.

Best Model Depth and Configuration: 4-layer linear neural network of node sizes (24, 48, 96, 192), Huber Loss, Adam Optimiser, Learning Rate of 0.001

$$L_\delta = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & if\ |(y - \hat{y})| < \delta \\ \delta((y - \hat{y}) - \frac{1}{2}\delta) & otherwise \end{cases}$$

Best Results: 55% win rate after 1000 episodes of training

Training time was too slow and convergence could not be reached in a reasonable time. I had already spent multiple weeks on this approach to no fruition. If I had more computational resources, I would train this model for much longer to see if true convergence can be reached.

5.2.2 Step 2: Analysing the Algorithms

The best performing algorithms were the hardcoded agent and Monte Carlo Tree Search, that produced high win rates (>80%). However, important observations for each strategy are:

- **Hardcoded Agent**: This agent is fast, but it is not efficient. It is only able to win the game if it is able to build a line of like pieces. If it cannot, it will return to a series of random moves that may/may not win the game.

- **Monte Carlo Tree Search**: MCTS rolls out and computes the reward from each board state but it is slow. It appears that it is not worth using at the start of the game, where a terminal state is quite distant from the current board position. Furthermore, a major problem with MCTS is the tree size, which grows exponentially with game progression. **This makes rolling out at each subsequent move slower than the previous rollout.**

Solution: Instead of keeping an extremely large tree, we record the result of each *state, action* pair in a Q-table, updated using Temporal Difference Learning and the Bellman equation. On the off chance that a past board state is encountered, the Q-table can be used to find the best corresponding action, instead of having to iterate through the entire tree. I call this the **QL-MCTS** algorithm, with inspiration from Wang et al. (2018) approach to Monte-Carlo Q-Learning. QL-MCTS works by:
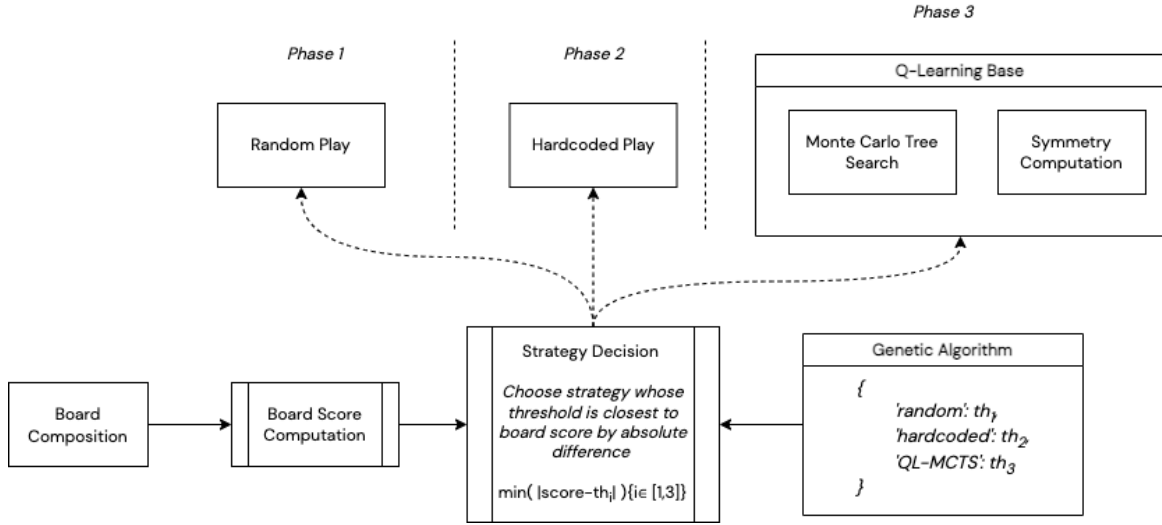
Figure 3: Hybrid Agent

- When training the Q-learning agent, use MCTS to find the best moves instead of using random in the epsilon-greedy policy.

- If the agent is called and a particular state-action combination is not present in the Q-table, go to MCTS to find the best move.

5.2.3 Step 3: Implementing the Hybrid Agent

Using the best performing algorithms, I created a hybrid agent that works in 3 phases. First, to get the game started, it will make random moves. After this, it will switch to a hardcoded strategy where it will attempt to computationally build lines of similar pieces. Finally, it will leverage the QL-MCTS algorithm to find the best moves and win the game. Since QL-MCTS is slow, it is kept as the final phase. This approach is shown in Figure 3, and is a balance between speed and efficiency.

The main question is when to switch between the algorithms. The intuition is that the switch depends on the change of the board composition. We represent this numerically through a board score, that is essentially a sum of couplets and triplets.

$$couples + 2 * triplets$$

The range of values for scores $\in [0, 16]$. We try to generate score thresholds to switch between the 3 strategies using a genetic algorithm. An example of a genome is shown in Figure 1.

We train the genetic algorithm for 1000 generations and a population size of 100, to find the best genome and submit these as the thresholds for the hybrid

---

Listing 1 Genome Example

```
{
    "random": 3,
    "hardcoded" : 5,
    "ql-mcts" : 8
}
```
---

agent.

Once the final thresholds are found, we find the strategy whose threshold has the smallest absolute difference with the current board score. The minimisation formula is:

$$\text{strategy} = \min_{i=1}^{3} |\text{threshold}_i - \text{board score}|$$

5.3   Results

After several iterations of the genetic algorithm, the best genome thresholds were found to be:

```
{
    'random': 2.090773081612301,
    'hardcoded': 3.790328881747581,
    'ql-mcts': 7.251997327518943
}
```

It is clear that the algorithm prefers to use the QL-MCTS (essentially MCTS) strategy extensively, as it almost always guarantees a win regardless of whether it is playing first or second. Random is entirely probabilistic and hardcoded has better chances only when it's the first player.

Tournament results are shown in Table 2, where each player is played against a random player for 10 tournaments of 10 games each (10 x 10 = 100 games).

Strategy	Win Rate	Comments
DQN	55%	Convergence not reached
QL-MCTS	84%	Slow, but can guarantee a win
Hardcoded	94%	Fast
Hybrid	78%	Heavily dependent on adjusting thresholds

Table 2: Results computed based on 10 tournaments of 10 games each (10 x 10 = 100 games) against a random player

5.4 Code of Hybrid Agent and Sub Players

This subsection covers the code of the final hybrid agent and the sub players it calls periodically.

5.4.1 Final Hybrid Player

The final hybrid player's driver code is below. It uses a genetic algorithm to find the best thresholds for switching between the 3 strategies. It uses crossover and conditional mutations to find the best genome from a limited population size.

Furthermore, to reduce the search space, we enforce the constraint that the threshold for random < hardcoded < MCTS, with the the intuition that the slowest but most powerful algorithm should be used last.

```python
'''
Genetic Algorithm for Quarto
'''
import sys
sys.path.insert(0, '..')

import tqdm
import random
import logging
import json
import itertools
from copy import deepcopy
from lib.players import Player, RandomPlayer
from quarto.objects import Quarto
from lib.scoring import score_board
from QLMCTS import QLearningPlayer
from Hardcoded.hardcoded import HardcodedPlayer

logging.basicConfig(level=logging.DEBUG)

class Genome:
    def __init__(self, thresholds, fitness):
        self.thresholds = thresholds
        self.fitness = fitness

    def set_fitness(self, fitness):
        self.fitness = fitness

    def set_thresholds(self, thresholds):
        self.thresholds = thresholds

    def toJSON(self):
        return {
            'thresholds': self.thresholds,
```

```python
35              'fitness': self.fitness
36          }
37
38
39  class FinalPlayer(Player):
40      '''
41      Final player uses genetic algorithm to decide between:
42      1. Hardcoded Strategy
43      2. Random Strategy
44      3. QL-MCTS
45      '''
46
47      def __init__(self, quarto: Quarto = None):
48          if quarto is None:
49              quarto = Quarto()
50          super().__init__(quarto)
51          self.ql_mcts = QLearningPlayer(quarto)
52          self.hardcoded = HardcodedPlayer(quarto)
53          self.random_player = RandomPlayer(quarto)
54          self.BOARD_SIDE = 4
55          self.GENOME_VAL_UPPER_BOUND = 16
56          self.GENOME_VAL_LOWER_BOUND = 0
57          self.thresholds = {
58              'random': 1.090773081612301,
59              'hardcoded': 2.790328881747581,
60              'ql-mcts': 8.251997327518943
61          }
62
63      def generate_population(self, population_size):
64          population = []
65          for i in range(population_size):
66              threshold = {}
67
68              # make sure that value for random < hardcoded < ql-mcts
69              threshold['random'] = random.random() * self.GENOME_VAL_UPPER_BOUND
70              # find random number between random and 15
71              threshold['hardcoded'] = threshold['random'] + \
72                  random.random() * (self.GENOME_VAL_UPPER_BOUND -
73                                      threshold['random'])
74
75              # find random number between hardcoded and 15
76              threshold['ql-mcts'] = threshold['hardcoded'] + \
77                  random.random() * (self.GENOME_VAL_UPPER_BOUND -
78                                      threshold['hardcoded'])
79
80              assert threshold['random'] < threshold['hardcoded'] <
81              ↪   threshold['ql-mcts']
82
83              population.append(Genome(threshold, 0))
84          return population
```

```python
84
85      def ensure_correct_ordering(self, new_thresholds):
86          if new_thresholds['random'] > new_thresholds['hardcoded']:
87              new_thresholds['random'], new_thresholds['hardcoded'] =
                ↪   new_thresholds['hardcoded'], new_thresholds['random']
88          if new_thresholds['hardcoded'] > new_thresholds['ql-mcts']:
89              new_thresholds['hardcoded'], new_thresholds['ql-mcts'] =
                ↪   new_thresholds['ql-mcts'], new_thresholds['hardcoded']
90          if new_thresholds['random'] > new_thresholds['hardcoded']:
91              new_thresholds['random'], new_thresholds['hardcoded'] =
                ↪   new_thresholds['hardcoded'], new_thresholds['random']
92          return new_thresholds
93
94      def crossover(self, genome1, genome2):
95          new_thresholds = {}
96          for key in genome1.thresholds:
97              new_thresholds[key] = random.choice(
98                  [genome1.thresholds[key], genome2.thresholds[key]])
99
100         # make sure that value for random < hardcoded < ql-mcts
101         new_thresholds = self.ensure_correct_ordering(new_thresholds)
102         return Genome(new_thresholds, 0)
103
104     def mutate(self, genome):
105         new_thresholds = {}
106         genome_thresholds = genome.thresholds
107         if random.random() < 0.4:
108             new_thresholds['random'] = random.random() * \
109                 self.GENOME_VAL_UPPER_BOUND
110             new_thresholds['hardcoded'] = random.choice(
111                 [genome_thresholds['random'], genome_thresholds['random'] +
112                     random.random() * (self.GENOME_VAL_UPPER_BOUND -
                    ↪   genome_thresholds['random'])])
113             new_thresholds['ql-mcts'] = random.choice(
114                 [genome_thresholds['hardcoded'], genome_thresholds['hardcoded'] +
115                     random.random() * (self.GENOME_VAL_UPPER_BOUND -
                    ↪   genome_thresholds['hardcoded'])])
116
117             new_thresholds = self.ensure_correct_ordering(new_thresholds)
118
119             assert new_thresholds['random'] < new_thresholds['hardcoded'] <
                ↪   new_thresholds['ql-mcts']
120
121             return Genome(new_thresholds, 0)
122         return genome
123
124     def evolve(self, num_generations=50):
125         self.population_size = 50
126         self.offspring_size = 10
127         population = self.generate_population(self.population_size)
```

```python
128
129            pbar = tqdm.tqdm(total=num_generations)
130            for gen in range(num_generations):
131                pbar.update(1)
132                logging.debug('Generation: {}'.format(gen))
133                offpsring = []
134                for i in range(self.offspring_size):
135                    parent1 = random.choice(population)
136                    parent2 = random.choice(population)
137                    child = self.crossover(parent1, parent2)
138                    child = self.mutate(child)
139                    child.fitness = self.play_game(child.thresholds, num_games=5)
140                    offpsring.append(child)
141                population += offpsring
142                population = sorted(
143                    population, key=lambda x: x.fitness,
                     ↪ reverse=True)[:self.population_size]

145                if gen % 5 == 0:
146                    logging.info('Saving population')
147                    with open('/Volumes/USB/population3.json', 'w') as f:
148                        json.dump([genome.toJSON() for genome in population], f)

150            # return the best genome's thresholds
151            return population[0].thresholds

153        def play_game(self, thresholds, num_games=10):
154            wins = 0
155            for game in range(num_games):
156                logging.debug('Game: {}'.format(game))
157                state = Quarto()
158                player = 0

160                # initialise with some random piece just to kickstart game
161                state.set_selected_piece(self.random_player.choose_piece(state, 0))
162                self.current_state = state

164                # python passes by reference
165                # agent will use the state, etc. to update the Q-table
166                # this function also wipes the MCTS tree
167                self.ql_mcts.clear_and_set_current_state(state)
168                self.hardcoded = HardcodedPlayer(state)

170                while True:
171                    # board score is the number of couples and triplets on the board
172                    # it is indicative of the change of the board state
173                    board_score = score_board(self.current_state)

175                    differences = [abs(board_score - thresholds[key])
176                                   for key in thresholds]
```

```
177                  min_diff = min(differences)
178                  index = differences.index(min_diff)
179                  key = list(thresholds.keys())[index]
180
181                  if player == 0:
182                      if key == 'random':
183                          logging.debug('random')
184                          # play randomly
185                          action = self.random_player.place_piece()
186                          next_piece = self.random_player.choose_piece()
187                          while
                         ↪  self.current_state.check_if_move_valid(self.current_state.get_s
                         ↪  action[0], action[1], next_piece) is False:
188                              action = self.random_player.place_piece()
189                              next_piece = self.random_player.choose_piece()
190                          self.current_state.select(
191                              self.current_state.get_selected_piece())
192                          self.current_state.place(action[0], action[1])
193                          self.current_state.set_selected_piece(next_piece)
194                          self.current_state.switch_player()
195                          player = 1 - player
196
197                      elif key == 'hardcoded':
198                          # play using hardcoded strategy
199                          self.previous_state = deepcopy(self.current_state)
200                          winning_piece, position =
                         ↪  self.hardcoded.hardcoded_strategy_get_move()
201                          next_piece =
                         ↪  self.hardcoded.hardcoded_strategy_get_piece()
202                          while
                         ↪  self.current_state.check_if_move_valid(self.current_state.get_s
                         ↪  position[0], position[1], next_piece) is False:
203                              winning_piece, position =
                             ↪  self.hardcoded.hardcoded_strategy_get_move(
204                                  self.current_state)
205                              next_piece =
                             ↪  self.hardcoded.hardcoded_strategy_get_piece(
206                                  self.current_state)
207                          self.current_state.select(state.get_selected_piece())
208                          self.current_state.place(position[0], position[1])
209                          self.current_state.set_selected_piece(next_piece)
210                          self.current_state.switch_player()
211                          player = 1 - player
212
213                      else:
214                          # play using QL-MCTS
215                          logging.debug('ql-mcts')
216                          self.ql_mcts.previous_state = deepcopy(
217                              self.current_state)
218                          action = self.ql_mcts.get_action(self.current_state)
```

```python
                        self.ql_mcts.previous_action = action
                        self.ql_mcts.current_state.select(
                            self.current_state.get_selected_piece())
                        self.ql_mcts.current_state.place(action[0], action[1])
                        self.ql_mcts.current_state.set_selected_piece(
                            action[2])
                        self.ql_mcts.current_state.switch_player()
                        player = 1 - player

                    else:
                        # opponent is random
                        action = self.random_player.place_piece()
                        next_piece = self.random_player.choose_piece()
                        while
                        ↪  self.current_state.check_if_move_valid(self.current_state.get_select
                        ↪  action[0], action[1], next_piece) is False:
                            action = self.random_player.place_piece()
                            next_piece = self.random_player.choose_piece()
                            # WARNING: very often stuck in this loop
                        self.current_state.select(
                            self.current_state.get_selected_piece())
                        self.current_state.place(action[0], action[1])
                        self.current_state.set_selected_piece(next_piece)
                        self.current_state.switch_player()
                        player = 1 - player

                    if self.current_state.check_is_game_over():
                        if 1 - self.current_state.check_winner() == 0:
                            print("Agent wins")
                            wins += 1
                            # TODO: QL reward update
                        else:
                            print("Player 2 wins")
                        break

        # fitness is the percentage of games won
        logging.debug(f"Win rate: {wins/num_games}")
        return wins/num_games

    def choose_piece(self):
        '''
        Choose piece for next player to place
        '''
        thresholds = self.thresholds

        # game is stored in parent
        self.current_state = self.get_game()

        board_score = score_board(self.current_state)
```

```python
267         differences = [abs(board_score - thresholds[key])
268                        for key in thresholds]
269         min_diff = min(differences)
270         index = differences.index(min_diff)
271         key = list(thresholds.keys())[index]
272
273         # python passes by reference
274         # agent will use the state, etc. to update the Q-table
275         # this function also wipes the MCTS tree
276         self.ql_mcts.clear_and_set_current_state(self.current_state)
277         self.hardcoded = HardcodedPlayer(self.current_state)
278
279         if key == 'random':
280             logging.debug('random')
281             # play randomly
282             action = self.random_player.place_piece()
283             next_piece = self.random_player.choose_piece()
284             while
             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_piece
             ↪ action[0], action[1], next_piece) is False:
285                 action = self.random_player.place_piece()
286                 next_piece = self.random_player.choose_piece()
287             return next_piece
288
289         elif key == 'hardcoded':
290             # play using hardcoded strategy
291             logging.debug('hardcoded')
292             self.previous_state = deepcopy(self.current_state)
293             # winning_piece, position =
             ↪ self.hardcoded.hardcoded_strategy_get_move()
294             next_piece = self.hardcoded.hardcoded_strategy_get_piece()
295             # while
             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_piec
             ↪ position[0], position[1], next_piece) is False:
296             #     winning_piece, position = self.hardcoded_strategy_get_move()
297             #     next_piece = self.hardcoded_strategy_get_piece()
298             return next_piece
299
300         else:
301             # play using QL-MCTS
302             logging.debug('ql-mcts')
303             self.ql_mcts.previous_state = deepcopy(
304                 self.current_state)
305             action = self.ql_mcts.get_action(self.current_state)
306             self.ql_mcts.previous_action = action
307             self.ql_mcts.current_state.select(
308                 self.current_state.get_selected_piece())
309             return action[2]
310
311     def place_piece(self):
```

```python
            # python passes by reference
            # agent will use the state, etc. to update the Q-table
            # this function also wipes the MCTS tree
            self.current_state = self.get_game()
            thresholds = self.thresholds

            # python passes by reference
            # agent will use the state, etc. to update the Q-table
            # this function also wipes the MCTS tree
            self.ql_mcts.clear_and_set_current_state(self.current_state)
            self.hardcoded = HardcodedPlayer(self.current_state)

        while True:
            # board score is the number of couples and triplets on the board
            # it is indicative of the change of the board state
            board_score = score_board(self.current_state)

            differences = [abs(board_score - thresholds[key])
                           for key in thresholds]
            min_diff = min(differences)
            index = differences.index(min_diff)
            key = list(thresholds.keys())[index]

            if key == 'random':
                logging.debug('random')
                # play randomly
                action = self.random_player.place_piece()
                next_piece = self.random_player.choose_piece()
                while
                    self.current_state.check_if_move_valid(self.current_state.get_selected_p
                    action[0], action[1], next_piece) is False:
                    action = self.random_player.place_piece()
                    next_piece = self.random_player.choose_piece()
                return action[0], action[1]

            elif key == 'hardcoded':
                # play using hardcoded strategy
                logging.debug('hardcoded')
                self.previous_state = deepcopy(self.current_state)
                winning_piece, position = self.hardcoded_strategy_get_move()
                # next_piece = self.hardcoded_strategy_get_piece()
                # while
                    self.current_state.check_if_move_valid(self.current_state.get_selected_
                    position[0], position[1], next_piece) is False:
                #     winning_piece, position =
                    self.hardcoded_strategy_get_move()
                #     next_piece = self.hardcoded_strategy_get_piece()
                return position[0], position[1]

            else:
```

```
357              # play using QL-MCTS
358              logging.debug('ql-mcts')
359              self.ql_mcts.previous_state = deepcopy(
360                  self.current_state)
361              action = self.ql_mcts.get_action(self.current_state)
362              self.ql_mcts.previous_action = action
363              return action[0], action[1]
364
365     def test_thresholds(self):
366         thresholds = {'random': 10000,
367                       'hardcoded': 3, 'ql-mcts': 1000}
368         win_rate = self.play_game(thresholds, num_games=10)
369         return win_rate
370
371 if __name__ == "__main__":
372     final_player = FinalPlayer()
373     # best_thresholds = final_player.evolve()
374     # print(best_thresholds)
375
376     average_win_rate = 0
377     for i in range(10):
378         win_rate = final_player.test_thresholds()
379         average_win_rate += win_rate
380     print(f"Average win rate: {average_win_rate}")
```

5.4.2 Code for Hardcoded Strategy

The strategy is outlined in this paper. I implement it in Python below.

```
1  '''
2  Hardcoded player for Quarto
3  Follows risky strategy from paper:
4
5  "Developing Strategic and Mathematical Thinking via Game Play:
6  Programming to Investigate a Risky Strategy for Quarto"
7  by Peter Rowlett
8  '''
9  from copy import deepcopy
10 import itertools
11 import logging
12 import random
13
14 from lib.players import Player
15 from quarto.objects import Quarto
16
17 import sys
18 sys.path.insert(0, '..')
19
20 class HardcodedPlayer(Player):
```

```python
    def __init__(self, quarto: Quarto = None):
        if quarto is None:
            quarto = Quarto()
        super().__init__(quarto)
        self.BOARD_SIDE = 4

    def check_if_winning_piece(self, state, piece):
        '''
        Simulate placing the piece on the board and check if the game is over
        '''

        for i in range(self.BOARD_SIDE):
            for j in range(self.BOARD_SIDE):
                if state.check_if_move_valid(piece, i, j, -100):
                    cloned_state = deepcopy(state)
                    cloned_state.select(piece)
                    cloned_state.place(i, j)

                    if cloned_state.check_is_game_over():
                        return True, [i, j]
        return False, None

    def hardcoded_strategy_get_piece(self):
        '''
        Returns a piece to be placed on the board
        '''
        state = self.get_game()

        possible_pieces = []
        for i in range(16):
            # check if the piece is a winning piece
            winning_piece, _ = self.check_if_winning_piece(state, i)
            if (not winning_piece) and (i not in
            ↪  list(itertools.chain.from_iterable(state.state_as_array()))) and
            ↪  (i != state.get_selected_piece()):
                possible_pieces.append(i)

        # if no pieces can be placed on board anymore (board full/game over),
        ↪  return -1
        if len(possible_pieces) == 0:
            # check if number of non-empty cells is 16
            if len([i for i in
            ↪  list(itertools.chain.from_iterable(state.state_as_array())) if i
            ↪  != -1]) == 16:
                return -1
            else:
                # there are possible pieces to be placed, but they are winning
                ↪  pieces/already in board
                on_board = list(itertools.chain.from_iterable(
                    state.state_as_array()))
```

```
65              not_on_board = list(set(range(16)) - set(on_board))
66              return random.choice(not_on_board)
67          else:
68              return random.choice(possible_pieces)
69
70      def choose_piece(self):
71          '''
72          Returns a piece to be placed on the board
73          '''
74          return self.hardcoded_strategy_get_piece()
75
76      def hardcoded_strategy_get_move(self, return_winning_piece_boolean=True):
77          #  1. Play the piece handed over by the opponent:
78          # (a) play a winning position if handed a winning piece;
79          # (b) otherwise, play to build a line of like pieces if possible;
80          # (c) otherwise, play randomly.
81          # 2. Hand a piece to the opponent:
82          # (a) avoid handing over a winning piece for your opponent to play;
83          # (b) otherwise, choose randomly.
84
85          state = self.get_game()
86
87          board = state.state_as_array()
88          selected_piece = state.get_selected_piece()
89          # check if the selected piece is a winning piece
90          winning_piece, position = self.check_if_winning_piece(
91              state, selected_piece)
92          if winning_piece:
93              return selected_piece, position
94
95          # check if the selected piece can be used to build a line of like pieces
96
97          row_1 = [[0, 0], [0, 1], [0, 2], [0, 3]]
98          # pieces in row 2
99          row_2 = [[1, 0], [1, 1], [1, 2], [1, 3]]
100         # pieces in row 3
101         row_3 = [[2, 0], [2, 1], [2, 2], [2, 3]]
102         # pieces in row 4
103         row_4 = [[3, 0], [3, 1], [3, 2], [3, 3]]
104
105         # pieces in column 1
106         col_1 = [[0, 0], [1, 0], [2, 0], [3, 0]]
107         # pieces in column 2
108         col_2 = [[0, 1], [1, 1], [2, 1], [3, 1]]
109         # pieces in column 3
110         col_3 = [[0, 2], [1, 2], [2, 2], [3, 2]]
111         # pieces in column 4
112         col_4 = [[0, 3], [1, 3], [2, 3], [3, 3]]
113
114         # pieces in diagonal 1
```

```
115         diag_1 = [[0, 0], [1, 1], [2, 2], [3, 3]]
116         # pieces in diagonal 2
117         diag_2 = [[0, 3], [1, 2], [2, 1], [3, 0]]
118
119         for line in [row_1, row_2, row_3, row_4, col_1, col_2, col_3, col_4,
            ↪  diag_1, diag_2]:
120             # check if the selected piece can be used to build a line of like
                ↪  pieces
121             characteristics = []
122             empty_rows = []
123             for el in line:
124                 x, y = el
125                 if board[x, y] != -1:
126                     piece = board[x][y]
127                     piece_char = state.get_piece_charchteristics(piece)
128                     characteristics.append(
129                         [piece_char.HIGH, piece_char.COLOURED, piece_char.SOLID,
                        ↪  piece_char.SQUARE])
130                 else:
131                     empty_rows.append(el)
132                     characteristics.append([-1, -1, -1, -1])
133
134         selected_piece_char = state.get_piece_charchteristics(
135             selected_piece)
136         selected_piece_char = [selected_piece_char.HIGH,
            ↪  selected_piece_char.COLOURED,
137                                selected_piece_char.SOLID,
                                ↪  selected_piece_char.SQUARE]
138
139         # check if characteristics has an empty row
140         if [-1, -1, -1, -1] in characteristics:
141             # count how many [-1, -1, -1, -1] are in characteristics
142             empty_indexes = [i for i, x in enumerate(
143                 characteristics) if x == [-1, -1, -1, -1]]
144
145             empty_rows_count = characteristics.count([-1, -1, -1, -1])
146             characteristics_copy = characteristics.copy()
147
148             # proceeding to check couplets and see if they can build
                ↪  triplets
149             # since 2 empty rows may be present and either could create a
                ↪  triplet, have to choose randomly later
150             potential_moves = []
151
152             for i, index in enumerate(empty_indexes):
153                 position = empty_rows[i]
154                 # insert the selected piece in the empty row
155                 # empty_piece_index = characteristics.index(
156                 #     [-1, -1, -1, -1])
157                 characteristics = characteristics_copy.copy()
```

```python
                        characteristics[index] = selected_piece_char

                        # check if any column has the same characteristics
                        col1 = [characteristics[0][0], characteristics[1][0],
                                characteristics[2][0], characteristics[3][0]]
                        col2 = [characteristics[0][1], characteristics[1][1],
                                characteristics[2][1], characteristics[3][1]]
                        col3 = [characteristics[0][2], characteristics[1][2],
                                characteristics[2][2], characteristics[3][2]]
                        col4 = [characteristics[0][3], characteristics[1][3],
                                characteristics[2][3], characteristics[3][3]]

                        col1 = [int(i) for i in col1]
                        col2 = [int(i) for i in col2]
                        col3 = [int(i) for i in col3]
                        col4 = [int(i) for i in col4]

                        # print(col1, col2, col3, col4)
                        def check_if_form_triplet(line):
                            # earlier we checked if we can complete a line
                            # here we check if we can form a triplet (one step away
                            ↪  from completing a line)
                            return line.count(1) == 3 or line.count(0) == 3

                        # if len(set(col1)) == 1 or len(set(col2)) == 1 or
                        ↪  len(set(col3)) == 1 or len(set(col4)) == 1:
                        if check_if_form_triplet(col1) or check_if_form_triplet(col2)
                        ↪  or check_if_form_triplet(col3) or
                        ↪  check_if_form_triplet(col4):
                            # this piece can be used to build a line of like pieces
                            logging.debug('playing to build a line of like pieces')
                            potential_moves.append(list(reversed(position)))

                    if len(potential_moves) >= 1:
                        if return_winning_piece_boolean:
                            # return True, list(reversed(empty_rows[-1]))
                            return True, random.choice(potential_moves)
                        else:
                            # move = list(reversed(empty_rows[-1]))
                            # move = list(reversed(position))
                            move = random.choice(potential_moves)
                            return move[0], move[1]

        # play randomly
        possible_moves = []
        for i in range(self.BOARD_SIDE):
            for j in range(self.BOARD_SIDE):
                for next_piece in range(16):
                    if state.check_if_move_valid(selected_piece, i, j,
                    ↪  next_piece):
```

```
203                        if return_winning_piece_boolean:
204                            possible_moves.append([False, [i, j]])
205                        else:
206                            possible_moves.append([i, j])
207
208            random_move = random.choice(possible_moves)
209            return random_move[0], random_move[1]
210
211            logging.debug(f"Selected piece: {selected_piece}")
212            logging.debug(f"Board: {board}")
213            logging.debug('no move found')
214
215        def place_piece(self):
216            '''
217            Above function sometimes necessary to return additional information
218            In game, first return value is not necessary
219            '''
220            return
                ↪ self.hardcoded_strategy_get_move(return_winning_piece_boolean=False)
```

5.4.3   Monte Carlo Tree Search

The implementation of MCTS and the rollout strategy is based on the minimal implementation here.

```
 1  from collections import defaultdict
 2  import copy
 3  import json
 4  import logging
 5  import math
 6  import pickle
 7  import random
 8  from threading import Thread
 9
10  import numpy as np
11  from lib.isomorphic import BoardTransforms
12  from lib.players import Player, RandomPlayer
13  from lib.utilities import Node, NodeDecoder, NodeEncoder
14
15  from quarto.objects import Quarto
16
17  logging.basicConfig(level=logging.INFO)
18
19
20  class MonteCarloTreeSearchEncoder(json.JSONEncoder):
21      def default(self, obj):
22          l = {
23              'Q': {k.hash_state(): v for k, v in obj.Q.items()},
24              'N': {k.hash_state(): v for k, v in obj.N.items()},
```

```
25
26              # children is a dictionary of nodes
27              'children': {k.hash_state(): [NodeEncoder().default(i) for i in v]
                ↪  for k, v in obj.children.items()},
28
29              # 'children': [NodeEncoder().default(child) for child in
                ↪  obj.children],
30              'epsilon': obj.epsilon,
31          }
32          return l
33
34      def encode(self, obj):
35          return super().encode(obj)
36
37      def load_json(self, filename):
38          with open(filename, 'r') as f:
39              return json.load(f, cls=MonteCarloTreeSearchDecoder)
40
41
42  class MonteCarloTreeSearchDecoder(json.JSONDecoder):
43      '''
44      Recreate MonteCarloTreeSearch object from JSON
45      '''
46
47      def __init__(self, *args, **kwargs):
48          json.JSONDecoder.__init__(
49              self, object_hook=self.object_hook, *args, **kwargs)
50
51      def object_hook(self, obj):
52          children = {}
53
54          for k, v in obj['children'].items():
55              children[Node(hashed_state=k)] = [
56                  NodeDecoder().object_hook(node) for node in v]
57
58          if 'Q' in obj:
59              return MonteCarloTreeSearch(
60                  Q={Node(hashed_state=k): v for k, v in obj['Q'].items()},
61                  N={Node(hashed_state=k): v for k, v in obj['N'].items()},
62                  children=children,
63                  epsilon=obj['epsilon'],
64              )
65          return obj
66
67
68  def decode_tree(tree):
69      return MonteCarloTreeSearchDecoder().object_hook(tree)
70
71
72  class MonteCarloTreeSearch(Player):
```

```python
      '''
      Solve using Monte Carlo Tree Search
      '''

      def __init__(self, board=Quarto(), epsilon=0.1, max_depth=1000, Q=None,
      ↪ N=None, children=None):
          self.epsilon = epsilon
          self.max_depth = max_depth
          if Q is None:
              self.Q = defaultdict(int)
          else:
              self.Q = defaultdict(int, Q)
          if N is None:
              self.N = defaultdict(int)
          else:
              self.N = defaultdict(int, N)
          if children is None:
              self.children = dict()
          else:
              self.children = children
          self.MAX_PIECES = 16
          self.BOARD_SIDE = 4
          self.board = board
          self.random_factor = 0
          self.decisions = 0
          super().__init__(board)

      def set_board(self, board):
          self.board = board

      def choose(self, node):
          '''
          Choose best successor of node (move)
          Returns the board itself
          '''
          def score(n):
              logging.debug(f"Before reading in choose {n}")
              if self.N[n] == 0:
                  return float('-inf')
              return self.Q[n] / self.N[n]

          # node is board Quarto
          node = Node(node)
          if node.is_terminal():
              logging.debug(node.board.state_as_array())
              raise RuntimeError("choose called on terminal node")

          # number of moves made in game
          self.decisions += 1
```

```python
122            for key in self.children:
123                if key == node:
124                    return max(self.children[key], key=score).board
125
126        self.random_factor += 1
127        if node not in self.children:
128            for key, value in self.children.items():
129                if BoardTransforms().compare_boards(node.board.state_as_array(),
     ↪  key.board.state_as_array()):
130                    if key in self.children:
131                        print("found in symmetry")
132                        return max(self.children[key], key=score).board
133
134            # number of times have to resort to random
135            rand_child = node.find_random_child()
136            # add to children
137            self.children[node] = [rand_child]
138            return rand_child.board
139
140        print("found in board")
141        return max(self.children[node], key=score).board
142
143    def choose_piece(self):
144        '''
145        Choose a piece to make the opponent place
146        '''
147        node = Node(board=self.board,
148                    selected_piece_index=self.board.get_selected_piece())
149
150        if node.is_terminal():
151            logging.debug(node.board.state_as_array())
152            raise RuntimeError("choose called on terminal node")
153
154        if node not in self.children:
155            # index -1 of tuple is next piece from a board
156            print("Random child")
157            return node.find_random_child()[-1]
158
159        def score(n):
160            logging.debug(f"Before reading in choose {n}")
161            if self.N[n] == 0:
162                return float('-inf')
163            return self.Q[n] / self.N[n]
164
165        return max(self.children[node], key=score)[-1]
166
167    def place_piece(self):
168        '''
169        Return position to place piece on board
170        '''
```

```python
        node = Node(board=self.board,
                    selected_piece_index=self.board.get_selected_piece())

        if node.is_terminal():
            logging.debug(node.board.state_as_array())
            raise RuntimeError("choose called on terminal node")

        # if node not in self.children:
        #     piece, x, y, next_piece = node.find_random_child().move
        #     # print("Random child")
        #     # print(piece, x, y, next_piece)
        #     return x, y, next_piece

        if node not in self.children:
            for key, value in self.children.items():
                if BoardTransforms().compare_boards(node.board.state_as_array(),
                   key.board.state_as_array()):
                    if key in self.children:
                        print("found in symmetry")
                        return max(self.children[key], key=score).board

            # number of times have to resort to random
            rand_child = node.find_random_child()
            print("Random child")
            # add to children
            return rand_child.board.move

        def score(n):
            logging.debug(f"Before reading in choose {n}")
            if self.N[n] == 0:
                return float('-inf')
            return self.Q[n] / self.N[n]

        # print("In place piece")
        # print(max(self.children[node], key=score).move)
        return max(self.children[node], key=score).move[1:]

    def do_rollout(self, board):
        '''
        Rollout from the node for one iteration
        '''
        logging.debug("Rollout")
        # if root node, there is no move
        node = Node(board, move=())
        path = self.select(node)
        leaf = path[-1]

        # expand a leaf only when necessary, i.e., only if I arrive at it during
        #    selection and if it has already been visited (self.N) but not yet
        #    expanded (self.children)
```

```python
            if leaf in self.N and leaf not in self.children:
                self.expand(leaf)

        reward = self.simulate(leaf)
        self.backpropagate(path, reward)

    def select(self, node):
        '''
        Select path to leaf node
        '''
        path = []
        while True:
            path.append(node)
            if node not in self.children or not self.children[node]:
                return path
            unexplored = self.children[node] - self.children.keys()
            if unexplored:
                n = unexplored.pop()
                path.append(n)
                return path
            node = self.uct_select(node)

    def expand(self, node):
        # logging.debug('Expanding')
        if node in self.children:
            return
        self.children[node] = node.find_children()
        # logging.debug('Children: ', self.children[node])

    def simulate(self, node):
        '''
        Returns reward for random simulation
        '''
        invert_reward = False
        while True:
            if node.is_terminal():
                reward = node.reward()

                return 1 - reward if invert_reward else reward
            node = node.find_random_child()
            # invert_reward = not invert_reward

    def backpropagate(self, path, reward):
        '''
        Backpropagate reward
        '''
        logging.debug('Backpropagating')
        for node in reversed(path):
            self.N[node] += 1
            self.Q[node] += reward
```

```python
                  # TODO: check if this is correct
                  reward = 1 - reward

      def uct_select(self, node):
          '''
          Select a child of node, balancing exploration & exploitation
          '''
          assert all(n in self.children for n in self.children[node])

          log_N_vertex = math.log(self.N[node])

          def uct(n):
              return self.Q[n] / self.N[n] + self.epsilon * math.sqrt(log_N_vertex
              ↪  / self.N[n])

          return max(self.children[node], key=uct)

      def test_win_rate(self, num_trials=10, rollouts=20):
          print("Testing win rate")
          agent_wins = 0
          opponent_wins = 0
          draws = 0
          for i in range(num_trials):
              board = Quarto()
              random_player = RandomPlayer(board)
              self.board = board
              board.set_selected_piece(random_player.choose_piece(board))
              while True:
                  # random player moves
                  chosen_location = random_player.place_piece(
                      board, board.get_selected_piece())
                  chosen_piece = random_player.choose_piece(board)
                  while not board.check_if_move_valid(board.get_selected_piece(),
                  ↪  chosen_location[0], chosen_location[1], chosen_piece):
                      chosen_location = random_player.place_piece(
                          board, board.get_selected_piece())
                      chosen_piece = random_player.choose_piece(board)
                  board.select(board.get_selected_piece())
                  board.place(chosen_location[0], chosen_location[1])
                  # setting the piece for the next player
                  board.set_selected_piece(chosen_piece)
                  board.switch_player()

                  if board.check_is_game_over():
                      if 1 - board.check_winner() == 0:
                          opponent_wins += 1
                      else:
                          draws += 1
                      break
                  # monte carlo tree search moves
```

```
316
317                     # make move with monte carlo tree search
318                     for _ in range(rollouts):
319                         self.do_rollout(board)
320                     board = self.choose(board)
321
322                     if board.check_is_game_over():
323                         # TODO: check if it's a draw
324                         if 1 - board.check_winner() == 1:
325                             agent_wins += 1
326                         else:
327                             draws += 1
328                         break
329                     # don't need to switch player because it's done in choose
330                     # random_player needs to do it because it is not done
     ↪        automatically
331
332         print(f"Agent wins: {agent_wins}/{i+1}")
333         print(f"Random factor ", self.random_factor / self.decisions)
334         self.random_factor = 0
335         self.decisions = 0
336
337     def train_engine(self, board, num_sims=200, save_format='json'):
338         '''
339         Train the model
340         '''
341         for i in range(num_sims):
342             board = Quarto()
343             random_player = RandomPlayer(board)
344             self.board = board
345             board.set_selected_piece(random_player.choose_piece(board))
346             logging.info(f"Iteration: {i} with tree size {len(self.children)}")
347             while True:
348                 # random player moves
349                 chosen_location = random_player.place_piece(
350                     board, board.get_selected_piece())
351                 chosen_piece = random_player.choose_piece(board)
352                 while not board.check_if_move_valid(board.get_selected_piece(),
     ↪        chosen_location[0], chosen_location[1], chosen_piece):
353                     chosen_location = random_player.place_piece(
354                         board, board.get_selected_piece())
355                     chosen_piece = random_player.choose_piece(board)
356                 board.select(board.get_selected_piece())
357                 board.place(chosen_location[0], chosen_location[1])
358                 # setting the piece for the next player
359                 board.set_selected_piece(chosen_piece)
360                 board.switch_player()
361
362                 if board.check_is_game_over():
363                     if 1 - board.check_winner() == 0:
```

```
                            logging.info("Random player won")
                    else:
                        logging.info("Draw")
                    break
            # monte carlo tree search moves

            # make move with monte carlo tree search
            for _ in range(20):
                self.do_rollout(board)
            board = self.choose(board)

            if board.check_is_game_over():
                # TODO: check if it's a draw
                if 1 - board.check_winner() == 1:
                    logging.info("Agent won")
                else:
                    logging.info("Draw")
                break
            # don't need to switch player because it's done in choose
            # random_player needs to do it because it is not done
            ↪ automatically

        if i % 2 == 0:
            # run a test to see if the agent is improving
            self.test_win_rate()

        # save progress every 10 iterations
        if i % 100 == 0:
            logging.debug("Saving progress")
            if save_format == 'json':
                self.save_progress_json('/Volumes/USB/progress3.json')
            else:
                self.save_progress_pickle('progress.pkl')

    def train(self):
        '''
        Train without multithreading
        '''
        self.train_engine(Quarto(), 100, 'json')

    def threaded_training(self, num_threads=1, save_format='json'):
        '''
        Train the model
        '''
        thread_pool = []

        for i in range(num_threads):
            t = Thread(target=self.train_engine, args=(Quarto(), 100, 'json'))
            t.start()
            thread_pool.append(t)
```

```python
        for t in thread_pool:
            t.join()

        # final save after training
        if save_format == 'json':
            self.save_progress_json('progress.json')
        else:
            self.save_progress_pickle('progress.pkl')

    def generate_future_probabilities(self, root: Node, node: Node):
        # 1 is the default value, but it can be changed to 0.5 or 0.1

        self.tau = 0.5
        if node not in self.children:
            self.do_rollout(root.board)

        probs = [self.N[child] / self.N[root]
                    for child in self.children[node]]

        probs = [p ** (1 / self.tau) for p in probs]

        probs = [p / sum(probs) for p in probs]

        return probs

    def save_progress_pickle(self, filename):
        with open(filename, 'wb') as f:
            pickle.dump(self, f)

    def save_progress_json(self, filename):
        with open(filename, 'w') as f:
            json.dump(self, f, cls=MonteCarloTreeSearchEncoder)

    def load_progress_json(self, filename):
        with open(filename, 'r') as f:
            return json.load(f, cls=MonteCarloTreeSearchDecoder)

    def load_progress(self, filename):
        with open(filename, 'rb') as f:
            return pickle.load(f)


if __name__ == "__main__":
    mcts = MonteCarloTreeSearch()
    # with open('/Volumes/USB/progress3.json', 'r') as f:
    #     mcts = decode_tree(json.load(f))
    #     logging.info("Loaded progress")
    logging.info("Starting training")
    mcts.train()
```

5.4.4 Q-Learning + MCTS

Here, I combine plain Q-Learning with an MCTS fallback, calling MCTS in the exploration hase and resorting to it in testing when a "state + action" pair cannot be found in the table.

```python
from collections import defaultdict
from copy import deepcopy
import itertools
import json
import logging
import math
import os
import random
import time

from MCTS import MonteCarloTreeSearch
from MCTS.mcts import decode_tree
from quarto.objects import Quarto
from lib.players import Player, RandomPlayer
from lib.isomorphic import BoardTransforms

import tqdm
logging.basicConfig(level=logging.DEBUG)


class QLearningPlayer(Player):
    def __init__(self, board: Quarto = Quarto(), epsilon=0.1, alpha=0.5,
    ↪ gamma=0.9, tree: MonteCarloTreeSearch = None):
        self.epsilon = epsilon
        self.alpha = alpha
        self.gamma = gamma
        self.board = board
        self.MAX_PIECES = 16
        self.BOARD_SIDE = 4
        self.Q = defaultdict(int)

        if tree is not None:
            # load the pre-initalised tree
            self.tree = tree
            self.tree.set_board(board)

        else:
            # load new tree
            self.tree = MonteCarloTreeSearch(board=board)

        super().__init__(board)

    def clear_and_set_current_state(self, state: Quarto):
        self.current_state = state
```

```python
44          self.tree = MonteCarloTreeSearch(board=state)

46      def reduce_normal_form(self, state: Quarto):
47          '''
48          Reduce the Quarto board to normal form (i.e. the board is symmetric)
49          '''
50          # NOT IMPLEMENTED for now, just return the board
51          return state

53      def hash_state_action(self, state: Quarto, action):
54          # reduce to normal form before saving to Q table
55          return state.board_to_string() + '||' + str(state.get_selected_piece()) +
        ↪  '||' + str(action)

57      def get_Q(self, state, action):
58          # check possible transforms first (really really slow)
59          for key, val in self.Q.items():
60              if BoardTransforms.compare_boards(state.state_as_array(),
                ↪  state.string_to_board(key.split('||')[0])):
61                  return val

63          if self.hash_state_action(state, action) not in self.Q:
64              # used to determine if state exists in Q table
65              # if None, then go to MCTS
66              return None

68          return self.Q[self.hash_state_action(state, action)]

70      def get_Q_for_state(self, state):
71          if self.hash_state_action(state, None) not in self.Q:
72              return None
73          return [i for i in self.Q if i.startswith(str(state))]

75      def set_Q(self, state, action, value):
76          self.Q[self.hash_state_action(state, action)] = value

78      def get_possible_actions(self, state: Quarto):
79          actions = []
80          for i in range(self.BOARD_SIDE):
81              for j in range(self.BOARD_SIDE):
82                  for piece in range(self.MAX_PIECES):
83                      if state.check_if_move_valid(self.board.get_selected_piece(),
                        ↪  i, j, piece):
84                          actions.append((i, j, piece))

86          return actions

88      def get_max_Q(self, state):
89          max_Q = -math.inf
90          for action in self.get_possible_actions(state):
```

```python
 91                if self.get_Q(state, action) is not None:
 92                    Q_val = self.get_Q(state, action)
 93                    max_Q = max(max_Q, self.get_Q(state, action))
 94            return max_Q
 95
 96        def check_if_winning_piece(self, state, piece):
 97            for i in range(self.BOARD_SIDE):
 98                for j in range(self.BOARD_SIDE):
 99                    if state.check_if_move_valid(piece, i, j, piece):
100                        cloned_state = deepcopy(state)
101                        cloned_state.select(piece)
102                        cloned_state.place(i, j)
103
104                        if cloned_state.check_is_game_over():
105                            print('WINNING PIECE FOUND')
106                            return True, [i, j]
107            return False, None
108
109        def hardcoded_strategy_get_move(self, state):
110            #  1. Play the piece handed over by the opponent:
111            # (a) play a winning position if handed a winning piece;
112            # (b) otherwise, play to build a line of like pieces if possible;
113            # (c) otherwise, play randomly.
114            # 2. Hand a piece to the opponent:
115            # (a) avoid handing over a winning piece for your opponent to play;
116            # (b) otherwise, choose randomly.
117
118            board = state.state_as_array()
119            selected_piece = state.get_selected_piece()
120            # check if the selected piece is a winning piece
121            winning_piece, position = self.check_if_winning_piece(
122                state, selected_piece)
123            if winning_piece:
124                return selected_piece, position
125
126            # check if the selected piece can be used to build a line of like pieces
127
128            row_1 = [[0, 0], [0, 1], [0, 2], [0, 3]]
129            # pieces in row 2
130            row_2 = [[1, 0], [1, 1], [1, 2], [1, 3]]
131            # pieces in row 3
132            row_3 = [[2, 0], [2, 1], [2, 2], [2, 3]]
133            # pieces in row 4
134            row_4 = [[3, 0], [3, 1], [3, 2], [3, 3]]
135
136            # pieces in column 1
137            col_1 = [[0, 0], [1, 0], [2, 0], [3, 0]]
138            # pieces in column 2
139            col_2 = [[0, 1], [1, 1], [2, 1], [3, 1]]
140            # pieces in column 3
```

```
141         col_3 = [[0, 2], [1, 2], [2, 2], [3, 2]]
142         # pieces in column 4
143         col_4 = [[0, 3], [1, 3], [2, 3], [3, 3]]
144
145         # pieces in diagonal 1
146         diag_1 = [[0, 0], [1, 1], [2, 2], [3, 3]]
147         # pieces in diagonal 2
148         diag_2 = [[0, 3], [1, 2], [2, 1], [3, 0]]
149
150         for line in [row_1, row_2, row_3, row_4, col_1, col_2, col_3, col_4,
        ↪  diag_1, diag_2]:
151             # check if the selected piece can be used to build a line of like
            ↪  pieces
152             characteristics = []
153             empty_rows = []
154             for el in line:
155                 x, y = el
156                 if board[x, y] != -1:
157                     piece = board[x][y]
158                     piece_char = state.get_piece_charachteristics(piece)
159                     characteristics.append(
160                         [piece_char.HIGH, piece_char.COLOURED, piece_char.SOLID,
                        ↪  piece_char.SQUARE])
161                 else:
162                     empty_rows.append(el)
163                     characteristics.append([-1, -1, -1, -1])
164
165             selected_piece_char = state.get_piece_charachteristics(
166                 selected_piece)
167             selected_piece_char = [selected_piece_char.HIGH,
            ↪  selected_piece_char.COLOURED,
168                                    selected_piece_char.SOLID,
                                   ↪  selected_piece_char.SQUARE]
169
170             # check if characteristics has an empty row
171             if [-1, -1, -1, -1] in characteristics:
172                 # insert the selected piece in the empty row
173                 empty_piece_index = characteristics.index(
174                     [-1, -1, -1, -1])
175                 characteristics[empty_piece_index] = selected_piece_char
176
177                 # check if any column has the same characteristics
178                 col1 = [characteristics[0][0], characteristics[1][0],
179                         characteristics[2][0], characteristics[3][0]]
180                 col2 = [characteristics[0][1], characteristics[1][1],
181                         characteristics[2][1], characteristics[3][1]]
182                 col3 = [characteristics[0][2], characteristics[1][2],
183                         characteristics[2][2], characteristics[3][2]]
184                 col4 = [characteristics[0][3], characteristics[1][3],
185                         characteristics[2][3], characteristics[3][3]]
```

125

```python
186
187                    col1 = [int(i) for i in col1]
188                    col2 = [int(i) for i in col2]
189                    col3 = [int(i) for i in col3]
190                    col4 = [int(i) for i in col4]
191
192                    if len(set(col1)) == 1 or len(set(col2)) == 1 or len(set(col3))
                        ↪  == 1 or len(set(col4)) == 1:
193                        # this piece can be used to build a line of like pieces
194                        logging.debug('playing to build a line of like pieces')
195                        return True, list(reversed(empty_rows[-1]))
196
197            # play randomly
198            for i in range(self.BOARD_SIDE):
199                for j in range(self.BOARD_SIDE):
200                    for next_piece in range(16):
201                        if state.check_if_move_valid(selected_piece, i, j,
                            ↪  next_piece):
202                            return False, [i, j]
203
204            print('returning nothing')
205            print(state.state_as_array())
206            print(state.get_selected_piece())
207
208        def hardcoded_strategy_get_piece(self, state):
209            possible_pieces = []
210            for i in range(16):
211                # check if the piece is a winning piece
212                winning_piece, _ = self.check_if_winning_piece(state, i)
213                if (not winning_piece) and (i not in
                    ↪  list(itertools.chain.from_iterable(state.state_as_array()))) and
                    ↪  (i != state.get_selected_piece()):
214                    possible_pieces.append(i)
215
216            return random.choice(possible_pieces)
217
218        def get_action(self, state, mode='training'):
219            '''
220            If state, action pair not in Q, go to Monte Carlo Tree Search to find
        ↪  best action
221            '''
222            if mode == 'training':
223                # exploration through epsilon greedy
224                # look for good moves through Monte Carlo Tree Search
225                if random.random() < self.epsilon:
226                    for i in range(20):
227                        self.tree.do_rollout(state)
228                    best_action = self.tree.place_piece()
229                    return best_action
230                else:
```

```python
                    # look in the q table for the best action
                    expected_score = 0
                    best_action = None
                    for action in self.get_possible_actions(state):
                        if self.get_Q(state, action) is not None and expected_score <
                        ↪   self.get_Q(state, action):
                            print('found in Q table')
                            expected_score = self.get_Q(state, action)
                            best_action = action
                    # go to Monte Carlo Tree Search if no suitable action found in Q
                    ↪   table
                    if best_action is None or expected_score == 0:
                        logging.debug(
                            'No suitable action found in Q table, going to Monte
                            ↪   Carlo Tree Search')
                        for i in range(20):
                            self.tree.do_rollout(state)
                        best_action = self.tree.place_piece()
                    else:
                        print('found in Q table')

                    return best_action
            else:
                # in test mode, use the Q table to find the best action
                # only go to Monte Carlo Tree Search if no suitable action found in Q
                ↪   table
                expected_score = 0
                best_action = None
                for action in self.get_possible_actions(state):
                    if self.get_Q(state, action) is not None and expected_score <
                    ↪   self.get_Q(state, action):
                        expected_score = self.get_Q(state, action)
                        best_action = action
                # go to Monte Carlo Tree Search if no suitable action found in Q
                ↪   table
                if best_action is None or expected_score == 0:
                    logging.debug(
                        'No suitable action found in Q table, going to Monte Carlo
                        ↪   Tree Search')
                    for i in range(50):
                        self.tree.do_rollout(state)
                    best_action = self.tree.place_piece()
                return best_action

    def update_Q(self, state, action, reward, next_state):
        Q_val = self.get_Q(state, action)
        if Q_val is None:
            Q_val = random.uniform(1.0, 0.01)
        self.set_Q(state, action, Q_val + self.alpha *
                    (reward + self.gamma * self.get_max_Q(next_state) - Q_val))
```

```python
274
275     def train(self, iterations=100):
276         # 1. Use the Q-function to initialize the value of each state-action
            ↪   pair, Q(s, a) = 0.
277         # automatically done through defaultdict
278
279         # Choose an action using MCTS
280         wins = 0
281         tries = 0
282         agent_decision_times = []
283
284         progress_bar = tqdm.tqdm(total=iterations)
285         for i in range(iterations):
286             board = Quarto()
287             self.board = board
288             random_player = RandomPlayer(board)
289             self.tree.set_board(board)
290             self.current_state = board
291             self.previous_state = None
292             self.previous_action = None
293             player = 1
294             self.current_state.switch_player()
295             selected_piece = random_player.choose_piece()
296             self.current_state.set_selected_piece(selected_piece)
297             while True:
298                 reward = 0
299                 if player == 0:
300                     # QL-MCTS moves here
301                     self.previous_state = deepcopy(self.current_state)
302                     logging.debug("Piece to place: ",
303                                     self.current_state.get_selected_piece())
304                     logging.debug("Board: ")
305                     logging.debug(self.current_state.state_as_array())
306                     time_start = time.time()
307                     action = self.get_action(self.current_state)
308                     self.previous_action = action
309                     time_end = time.time()
310                     agent_decision_times.append(time_end - time_start)
311                     self.current_state.select(selected_piece)
312                     self.current_state.place(action[0], action[1])
313                     self.current_state.set_selected_piece(action[2])
314                     self.current_state.switch_player()
315                     player = 1 - player
316
317                 else:
318                     # Random moves here
319                     action = random_player.place_piece()
320                     next_piece = random_player.choose_piece()
```

```python
                        while
                        ↪  self.board.check_if_move_valid(self.board.get_selected_piece(),
                        ↪  action[0], action[1], next_piece) is False:
                            action = random_player.place_piece()
                            next_piece = random_player.choose_piece()
                        self.current_state.select(
                            self.current_state.get_selected_piece())
                        self.current_state.place(action[0], action[1])
                        self.current_state.set_selected_piece(next_piece)
                        self.current_state.switch_player()
                        player = 1 - player

                if self.current_state.check_is_game_over():
                    if 1 - self.current_state.check_winner() == 1:
                        logging.info('QL-MCTS won')
                        reward = 1
                        wins += 1
                    else:
                        logging.info('Random won')
                        reward = -1
                    self.update_Q(self.previous_state, self.previous_action,
                                  reward, self.current_state)
                    break
                else:
                    if self.previous_state is not None:
                        self.update_Q(
                            self.previous_state, self.previous_action, reward,
                            ↪  self.current_state)

            tries += 1
            if i % 10 == 0:
                logging.info(f'Iteration {i}')
                logging.info(f'Wins: {wins}')
                logging.info(f'Tries: {tries}')
                logging.info(f'Win rate: {wins/tries}')
                wins = 0
                tries = 0

            # OPTION 1: clear the tree every time
            self.tree = MonteCarloTreeSearch(board=self.board)

            # OPTION 2: if average agent decision time is too long, clear the
            ↪  MCTS tree
            # if sum(agent_decision_times) / len(agent_decision_times) > 5:
            #     self.tree = MonteCarloTreeSearch(board=self.board)
            #     agent_decision_times = []

            progress_bar.update(1)
```

```
367  if __name__ == '__main__':
368      # load tree with MonteCarloSearchDecoder
369      with open('progress.json', 'r') as f:
370          tree = decode_tree(json.load(f))
371      qplayer = QLearningPlayer()
372      qplayer.train(10)
```

5.5 Utility Functions

```
1   def score_board(state: Quarto):
2       positions = {
3           'couples': 0,
4           'triplets': 0,
5       }
6
7       board = state.state_as_array()
8
9       # Array for checking if a certain row already contains a triplet
10      row_done = [False, False, False, False]
11
12      # Check all the rows
13
14      for i in range(state.BOARD_SIDE):
15          if board[i, 0] != -1 and board[i, 1] != -1 and board[i, 2] != -1 and
           ↪ board[i, 3] == -1:
16              piece_1 = state.get_piece_charachteristics(board[i, 0])
17              piece_2 = state.get_piece_charachteristics(board[i, 1])
18              piece_3 = state.get_piece_charachteristics(board[i, 2])
19              if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
               ↪ piece_1.HIGH == True:
20                  positions['triplets'] += 1
21                  row_done[i] = True
22              if piece_1.HIGH == piece_2.COLOURED and piece_3.COLOURED ==
               ↪ piece_2.COLOURED and piece_1.COLOURED == True:
23                  positions['triplets'] += 1
24                  row_done[i] = True
25              if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
               ↪ piece_2.SQUARE and piece_1.SQUARE == True:
26                  positions['triplets'] += 1
27                  row_done[i] = True
28              if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
               ↪ and piece_1.SOLID == True:
29                  positions['triplets'] += 1
30                  row_done[i] = True
31          if board[i, 0] == -1 and board[i, 1] != -1 and board[i, 2] != -1 and
           ↪ board[i, 3] != -1:
32              piece_1 = state.get_piece_charachteristics(board[i, 1])
33              piece_2 = state.get_piece_charachteristics(board[i, 2])
```

```
34          piece_3 = state.get_piece_charachteristics(board[i, 3])
35          if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
       ↪    piece_1.HIGH == True:
36              positions['triplets'] += 1
37              row_done[i] = True
38          if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
       ↪    piece_2.COLOURED and piece_1.COLOURED == True:
39              positions['triplets'] += 1
40              row_done[i] = True
41          if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
       ↪    piece_2.SQUARE and piece_1.SQUARE == True:
42              positions['triplets'] += 1
43              row_done[i] = True
44          if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
       ↪    and piece_1.SOLID == True:
45              positions['triplets'] += 1
46              row_done[i] = True
47      if board[i, 0] != -1 and board[i, 1] != -1 and board[i, 2] == -1 and
   ↪    board[i, 3] != -1:
48          piece_1 = state.get_piece_charachteristics(board[i, 0])
49          piece_2 = state.get_piece_charachteristics(board[i, 1])
50          piece_3 = state.get_piece_charachteristics(board[i, 3])
51          if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
       ↪    piece_1.HIGH == True:
52              positions['triplets'] += 1
53              row_done[i] = True
54          if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
       ↪    piece_2.COLOURED and piece_1.COLOURED == True:
55              positions['triplets'] += 1
56              row_done[i] = True
57          if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
       ↪    piece_2.SQUARE and piece_1.SQUARE == True:
58              positions['triplets'] += 1
59              row_done[i] = True
60          if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
       ↪    and piece_1.SOLID == True:
61              positions['triplets'] += 1
62              row_done[i] = True
63      if board[i, 0] != -1 and board[i, 1] == -1 and board[i, 2] != -1 and
   ↪    board[i, 3] != -1:
64          piece_1 = state.get_piece_charachteristics(board[i, 0])
65          piece_2 = state.get_piece_charachteristics(board[i, 2])
66          piece_3 = state.get_piece_charachteristics(board[i, 3])
67          if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
       ↪    piece_1.HIGH == True:
68              positions['triplets'] += 1
69              row_done[i] = True
70          if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
       ↪    piece_2.COLOURED and piece_1.COLOURED == True:
71              positions['triplets'] += 1
```

```
72                    row_done[i] = True
73                  if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
      ↪ piece_2.SQUARE and piece_1.SQUARE == True:
74                      positions['triplets'] += 1
75                      row_done[i] = True
76                  if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
      ↪ and piece_1.SOLID == True:
77                      positions['triplets'] += 1
78                      row_done[i] = True
79              # Before checking a row for couples, control the boolean flag
80              if not row_done[i] and board[i, 0] != -1 and board[i, 1] != -1 and
      ↪ board[i, 2] == -1 and board[i, 3] == -1:
81                  piece_1 = state.get_piece_charachteristics(board[i, 0])
82                  piece_2 = state.get_piece_charachteristics(board[i, 1])
83                  if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
84                      positions['couples'] += 1
85                  if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
86                      positions['couples'] += 1
87                  if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
88                      positions['couples'] += 1
89                  if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
90                      positions['couples'] += 1
91              if not row_done[i] and board[i, 0] == -1 and board[i, 1] != -1 and
      ↪ board[i, 2] != -1 and board[i, 3] == -1:
92                  piece_1 = state.get_piece_charachteristics(board[i, 1])
93                  piece_2 = state.get_piece_charachteristics(board[i, 2])
94                  if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
95                      positions['couples'] += 1
96                  if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
97                      positions['couples'] += 1
98                  if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
99                      positions['couples'] += 1
100                 if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
101                     positions['couples'] += 1
102             if not row_done[i] and board[i, 0] == -1 and board[i, 1] == -1 and
      ↪ board[i, 2] != -1 and board[i, 3] != -1:
103                 piece_1 = state.get_piece_charachteristics(board[i, 2])
104                 piece_2 = state.get_piece_charachteristics(board[i, 3])
105                 if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
106                     positions['couples'] += 1
107                 if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
108                     positions['couples'] += 1
109                 if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
110                     positions['couples'] += 1
111                 if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
112                     positions['couples'] += 1
113
114         # Array for checking if a certain column already contains a triplet
115         column_done = [False, False, False, False]
116
```

```python
        # Check all the columns

    for i in range(state.BOARD_SIDE):
        if board[0, i] != -1 and board[1, i] != -1 and board[2, i] != -1 and
        ↪  board[3, i] == -1:
            piece_1 = state.get_piece_charachteristics(board[0, i])
            piece_2 = state.get_piece_charachteristics(board[1, i])
            piece_3 = state.get_piece_charachteristics(board[2, i])
            if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↪  piece_1.HIGH == True:
                positions['triplets'] += 1
                column_done[i] = True
            if piece_1.HIGH == piece_2.COLOURED and piece_3.COLOURED ==
            ↪  piece_2.COLOURED and piece_1.COLOURED == True:
                positions['triplets'] += 1
                column_done[i] = True
            if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
            ↪  piece_2.SQUARE and piece_1.SQUARE == True:
                positions['triplets'] += 1
                column_done[i] = True
            if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
            ↪  and piece_1.SOLID == True:
                positions['triplets'] += 1
                column_done[i] = True
        if board[0, i] == -1 and board[1, i] != -1 and board[2, i] != -1 and
        ↪  board[3, i] != -1:
            piece_1 = state.get_piece_charachteristics(board[1, i])
            piece_2 = state.get_piece_charachteristics(board[2, i])
            piece_3 = state.get_piece_charachteristics(board[3, i])
            if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↪  piece_1.HIGH == True:
                positions['triplets'] += 1
                column_done[i] = True
            if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
            ↪  piece_2.COLOURED and piece_1.COLOURED == True:
                positions['triplets'] += 1
                column_done[i] = True
            if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
            ↪  piece_2.SQUARE and piece_1.SQUARE == True:
                positions['triplets'] += 1
                column_done[i] = True
            if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
            ↪  and piece_1.SOLID == True:
                positions['triplets'] += 1
                column_done[i] = True
        if board[0, i] != -1 and board[1, i] == -1 and board[2, i] != -1 and
        ↪  board[3, i] != -1:
            piece_1 = state.get_piece_charachteristics(board[0, i])
            piece_2 = state.get_piece_charachteristics(board[2, i])
            piece_3 = state.get_piece_charachteristics(board[3, i])
```

```
156        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
   ↪   piece_1.HIGH == True:
157            positions['triplets'] += 1
158            column_done[i] = True
159        if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
   ↪   piece_2.COLOURED and piece_1.COLOURED == True:
160            positions['triplets'] += 1
161            column_done[i] = True
162        if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
   ↪   piece_2.SQUARE and piece_1.SQUARE == True:
163            positions['triplets'] += 1
164            column_done[i] = True
165        if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
   ↪   and piece_1.SOLID == True:
166            positions['triplets'] += 1
167            column_done[i] = True
168    if board[0, i] != -1 and board[1, i] != -1 and board[2, i] == -1 and
   ↪   board[3, i] != -1:
169        piece_1 = state.get_piece_charachteristics(board[0, i])
170        piece_2 = state.get_piece_charachteristics(board[1, i])
171        piece_3 = state.get_piece_charachteristics(board[3, i])
172        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
   ↪   piece_1.HIGH == True:
173            positions['triplets'] += 1
174            column_done[i] = True
175        if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
   ↪   piece_2.COLOURED and piece_1.COLOURED == True:
176            positions['triplets'] += 1
177            column_done[i] = True
178        if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
   ↪   piece_2.SQUARE and piece_1.SQUARE == True:
179            positions['triplets'] += 1
180            column_done[i] = True
181        if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
   ↪   and piece_1.SOLID == True:
182            positions['triplets'] += 1
183            column_done[i] = True
184    # Before checking a column for couples, control the boolean flag
185    if not column_done[i] and board[0, i] != -1 and board[1, i] != -1 and
   ↪   board[2, i] == -1 and board[3, i] == -1:
186        piece_1 = state.get_piece_charachteristics(board[0, i])
187        piece_2 = state.get_piece_charachteristics(board[1, i])
188        if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
189            positions['couples'] += 1
190        if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
191            positions['couples'] += 1
192        if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
193            positions['couples'] += 1
194        if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
195            positions['couples'] += 1
```

```python
            if not column_done[i] and board[0, i] == -1 and board[1, i] != -1 and
            ↪  board[2, i] != -1 and board[3, i] == -1:
                piece_1 = state.get_piece_charachteristics(board[1, i])
                piece_2 = state.get_piece_charachteristics(board[2, i])
                if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
                    positions['couples'] += 1
                if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
                    positions['couples'] += 1
                if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
                    positions['couples'] += 1
                if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
                    positions['couples'] += 1
            if not column_done[i] and board[0, i] == -1 and board[1, i] == -1 and
            ↪  board[2, i] != -1 and board[3, i] != -1:
                piece_1 = state.get_piece_charachteristics(board[2, i])
                piece_2 = state.get_piece_charachteristics(board[3, i])
                if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
                    positions['couples'] += 1
                if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
                    positions['couples'] += 1
                if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
                    positions['couples'] += 1
                if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
                    positions['couples'] += 1

        # Array for checking if a certain diagonal already contains a triplet
        diagonal_done = [False, False]

        # Check first diagonal

        if board[0, 0] != -1 and board[1, 1] != -1 and board[2, 2] != -1 and board[3,
        ↪  3] == -1:
            piece_1 = state.get_piece_charachteristics(board[0, 0])
            piece_2 = state.get_piece_charachteristics(board[1, 1])
            piece_3 = state.get_piece_charachteristics(board[2, 2])
            if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↪  piece_1.HIGH == True:
                positions['triplets'] += 1
                diagonal_done[0] = True
            if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
            ↪  piece_2.COLOURED and piece_1.COLOURED == True:
                positions['triplets'] += 1
                diagonal_done[0] = True
            if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
            ↪  and piece_1.SQUARE == True:
                positions['triplets'] += 1
                diagonal_done[0] = True
            if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
            ↪  piece_1.SOLID == True:
                positions['triplets'] += 1
```

```python
                    diagonal_done[0] = True
    if board[0, 0] == -1 and board[1, 1] != -1 and board[2, 2] != -1 and board[3,
    ↪    3] != -1:
        piece_1 = state.get_piece_charachteristics(board[1, 1])
        piece_2 = state.get_piece_charachteristics(board[2, 2])
        piece_3 = state.get_piece_charachteristics(board[3, 3])
        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
        ↪    piece_1.HIGH == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
        if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
        ↪    piece_2.COLOURED and piece_1.COLOURED == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
        if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
        ↪    and piece_1.SQUARE == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
        if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
        ↪    piece_1.SOLID == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
    if board[0, 0] != -1 and board[1, 1] == -1 and board[2, 2] != -1 and board[3,
    ↪    3] != -1:
        piece_1 = state.get_piece_charachteristics(board[0, 0])
        piece_2 = state.get_piece_charachteristics(board[2, 2])
        piece_3 = state.get_piece_charachteristics(board[3, 3])
        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
        ↪    piece_1.HIGH == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
        if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
        ↪    piece_2.COLOURED and piece_1.COLOURED == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
        if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
        ↪    and piece_1.SQUARE == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
        if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
        ↪    piece_1.SOLID == True:
            positions['triplets'] += 1
            diagonal_done[0] = True
    if board[0, 0] != -1 and board[1, 1] != -1 and board[2, 2] == -1 and board[3,
    ↪    3] != -1:
        piece_1 = state.get_piece_charachteristics(board[0, 0])
        piece_2 = state.get_piece_charachteristics(board[1, 1])
        piece_3 = state.get_piece_charachteristics(board[3, 3])
        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
        ↪    piece_1.HIGH == True:
```

```
277                positions['triplets'] += 1
278                diagonal_done[0] = True
279            if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
      ↪    piece_2.COLOURED and piece_1.COLOURED == True:
280                positions['triplets'] += 1
281                diagonal_done[0] = True
282            if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
      ↪    and piece_1.SQUARE == True:
283                positions['triplets'] += 1
284                diagonal_done[0] = True
285            if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
      ↪    piece_1.SOLID == True:
286                positions['triplets'] += 1
287                diagonal_done[0] = True
288        # Before checking a diagonal for couples, control the boolean flag
289        if not diagonal_done[0] and board[0, 0] != -1 and board[1, 1] != -1 and
    ↪    board[2, 2] == -1 and board[3, 3] == -1:
290            piece_1 = state.get_piece_charachteristics(board[0, 0])
291            piece_2 = state.get_piece_charachteristics(board[1, 1])
292            if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
293                positions['couples'] += 1
294            if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
295                positions['couples'] += 1
296            if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
297                positions['couples'] += 1
298            if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
299                positions['couples'] += 1
300        if not diagonal_done[0] and board[0, 0] == -1 and board[1, 1] != -1 and
    ↪    board[2, 2] != -1 and board[3, 3] == -1:
301            piece_1 = state.get_piece_charachteristics(board[1, 1])
302            piece_2 = state.get_piece_charachteristics(board[2, 2])
303            if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
304                positions['couples'] += 1
305            if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
306                positions['couples'] += 1
307            if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
308                positions['couples'] += 1
309            if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
310                positions['couples'] += 1
311        if not diagonal_done[0] and board[0, 0] == -1 and board[1, 1] == -1 and
    ↪    board[2, 2] != -1 and board[3, 3] != -1:
312            piece_1 = state.get_piece_charachteristics(board[2, 2])
313            piece_2 = state.get_piece_charachteristics(board[3, 3])
314            if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
315                positions['couples'] += 1
316            if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
317                positions['couples'] += 1
318            if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
319                positions['couples'] += 1
320            if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
```

```
321             positions['couples'] += 1
322
323         # Check second diagonal
324
325         if board[0, 3] != -1 and board[1, 2] != -1 and board[2, 1] != -1 and board[3,
        ↪  0] == -1:
326             piece_1 = state.get_piece_charachteristics(board[0, 3])
327             piece_2 = state.get_piece_charachteristics(board[1, 2])
328             piece_3 = state.get_piece_charachteristics(board[2, 1])
329             if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↪  piece_1.HIGH == True:
330                 positions['triplets'] += 1
331                 diagonal_done[1] = True
332             if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
            ↪  piece_2.COLOURED and piece_1.COLOURED == True:
333                 positions['triplets'] += 1
334                 diagonal_done[1] = True
335             if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
            ↪  and piece_1.SQUARE == True:
336                 positions['triplets'] += 1
337                 diagonal_done[1] = True
338             if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
            ↪  piece_1.SOLID == True:
339                 positions['triplets'] += 1
340                 diagonal_done[1] = True
341         if board[0, 3] == -1 and board[1, 2] != -1 and board[2, 1] != -1 and board[3,
        ↪  0] != -1:
342             piece_1 = state.get_piece_charachteristics(board[1, 2])
343             piece_2 = state.get_piece_charachteristics(board[2, 1])
344             piece_3 = state.get_piece_charachteristics(board[3, 0])
345             if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↪  piece_1.HIGH == True:
346                 positions['triplets'] += 1
347                 diagonal_done[1] = True
348             if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
            ↪  piece_2.COLOURED and piece_1.COLOURED == True:
349                 positions['triplets'] += 1
350                 diagonal_done[1] = True
351             if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
            ↪  and piece_1.SQUARE == True:
352                 positions['triplets'] += 1
353                 diagonal_done[1] = True
354             if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
            ↪  piece_1.SOLID == True:
355                 positions['triplets'] += 1
356                 diagonal_done[1] = True
357         if board[0, 3] != -1 and board[1, 2] != -1 and board[2, 1] == -1 and board[3,
        ↪  0] != -1:
358             piece_1 = state.get_piece_charachteristics(board[0, 3])
359             piece_2 = state.get_piece_charachteristics(board[1, 2])
```

```python
        piece_3 = state.get_piece_charachteristics(board[3, 0])
        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
        ↪  piece_1.HIGH == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
        if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
        ↪  piece_2.COLOURED and piece_1.COLOURED == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
        if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
        ↪  and piece_1.SQUARE == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
        if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
        ↪  piece_1.SOLID == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
    if board[0, 3] != -1 and board[1, 2] == -1 and board[2, 1] != -1 and board[3,
    ↪  0] != -1:
        piece_1 = state.get_piece_charachteristics(board[0, 3])
        piece_2 = state.get_piece_charachteristics(board[2, 1])
        piece_3 = state.get_piece_charachteristics(board[3, 0])
        if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
        ↪  piece_1.HIGH == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
        if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
        ↪  piece_2.COLOURED and piece_1.COLOURED == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
        if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
        ↪  and piece_1.SQUARE == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
        if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
        ↪  piece_1.SOLID == True:
            positions['triplets'] += 1
            diagonal_done[1] = True
    # Before checking a diagonal for couples, control the boolean flag
    if not diagonal_done[1] and board[0, 3] != -1 and board[1, 2] != -1 and
    ↪  board[2, 1] == -1 and board[3, 0] == -1:
        piece_1 = state.get_piece_charachteristics(board[0, 3])
        piece_2 = state.get_piece_charachteristics(board[1, 2])
        if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
            positions['couples'] += 1
        if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
            positions['couples'] += 1
        if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
            positions['couples'] += 1
        if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
```

```
400              positions['couples'] += 1
401       if not diagonal_done[1] and board[0, 3] == -1 and board[1, 2] != -1 and
    ↪ board[2, 1] != -1 and board[3, 0] == -1:
402           piece_1 = state.get_piece_charachteristics(board[1, 2])
403           piece_2 = state.get_piece_charachteristics(board[2, 1])
404           if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
405               positions['couples'] += 1
406           if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
407               positions['couples'] += 1
408           if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
409               positions['couples'] += 1
410           if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
411               positions['couples'] += 1
412       if not diagonal_done[1] and board[0, 0] == -1 and board[1, 2] == -1 and
    ↪ board[2, 1] != -1 and board[3, 0] != -1:
413           piece_1 = state.get_piece_charachteristics(board[2, 1])
414           piece_2 = state.get_piece_charachteristics(board[3, 0])
415           if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
416               positions['couples'] += 1
417           if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
418               positions['couples'] += 1
419           if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
420               positions['couples'] += 1
421           if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
422               positions['couples'] += 1
423
424       return positions['couples']+2*positions['triplets']
```

5.6 Code for Unsuccessful Players

5.6.1 Deep Q-Network

```
1  """
2  In this file, I build a Deep Q-Network to play Quarto.
3  """
4  import sys
5
6  sys.path.insert(0, '..')
7
8  from quarto.gym_environment import QuartoScape
9  from collections import deque
10 import logging
11 import os
12 import random
13 from typing import Any
14 import gym
15 import numpy as np
16 import tensorflow as tf
```

```python
from lib.players import RandomPlayer
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.initializers import HeUniform

from quarto.objects import Quarto

env = QuartoScape()


def test(agent):
    dq_wins = 0
    for round in range(100):
        game = Quarto()
        agent.set_game(game)
        game.set_players((RandomPlayer(game), agent))
        winner = game.run()
        if winner == 1:
            dq_wins += 1
        # logging.warning(f"main: Winner: player {winner}")
    logging.warning(f"main: DQ wins: {dq_wins}")


class DQNAgent:
    '''Play Quarto using a Deep Q-Network'''

    def __init__(self, env=env, game=None):
        self.env = env
        # main model updated every x steps
        self.model = self._build_model()
        # target model updated every y steps
        self.target_model = self._build_model()
        self.gamma = 0.618
        self.min_replay_size = 500
        self.lr = 0.7
        self.epsilon = 0.8
        if game is not None:
            self.env.game = game

        if os.path.exists('model.h5'):
            # print('Loading model')
            self.model = tf.keras.models.load_model('model.h5')

    def set_game(self, game):
        self.env.game = game

    def get_all_actions(self):
        '''
        Return tuples from (0, 0, 0) to (3, 3, 15)
```

```python
            Element 1 is position x
            Element 2 is position y
            Element 3 is piece chosen for next player
            '''
            tuples = []
            for i in range(0, 4):
                for j in range(0, 4):
                    for k in range(0, 16):
                        tuples.append((i, j, k))
            return tuples

    def _build_model(self):
        '''
        Architecture of network:
        Input nodes are the state of the board
        Output nodes are the Q-values for each potential action (each output node
    is an action)
        An action is made up of (x, y, piece chosen for next player)
        There are 16 * 16 * 16 possible actions and the mapping is found in
    get_all_actions()
        '''
        model = Sequential()
        initializer = HeUniform()
        model.add(Dense(
            12, input_dim=self.env.observation_space.shape[0], activation='relu',
             kernel_initializer=initializer))
        model.add(Dense(24, activation='relu', kernel_initializer=initializer))
        model.add(Dense(48, activation='relu', kernel_initializer=initializer))
        model.add(Dense(96, activation='relu', kernel_initializer=initializer))
        model.add(Dense(192, activation='relu',
                    kernel_initializer=initializer))
        model.add(Dense(4 * 4 * 16, activation='linear',
                    kernel_initializer=initializer))
        model.compile(loss=tf.keras.losses.Huber(), metrics=[
                        'mae', 'mse'], optimizer=Adam(learning_rate=0.001))
        return model

    def build_conv_model(self):
        model = Sequential()
        model.add(Conv2D(32, (3, 3), input_shape=(4, 4, 4), activation='relu'))
        model.add(MaxPooling2D(pool_size=(2, 2)))
        model.add(Flatten())
        model.add(Dense(16, activation='relu'))
        model.add(Dense(4 * 4 * 16, activation='linear'))
        model.compile(loss='mse', metrics=[
                        'accuracy'], optimizer=Adam(learning_rate=0.001))
        return model

    def get_position(self, element, list):
        if element in list:
```

```python
114                     return list.index(element)
115             else:
116                 return -1
117
118         def make_prediction(self, state, chosen_piece=None):
119             '''Make a prediction using the network'''
120             # prediction X is the position of the single 1 in the state
121             pred_X = [self.get_position(i, list(state.flatten()))
122                         for i in range(0, 16)]
123             pred_X.append(chosen_piece)
124             return self.model.predict(np.array([pred_X]), verbose=0)[0]
125
126         def decay_lr(self, lr, decay_rate, decay_step):
127             return lr * (1 / (1 + decay_rate * decay_step))
128
129         def abbellire(self, state, chosen_piece):
130             '''
131             Beautify the state for network input
132             When in Italy, do as the Italians do
133             '''
134             X = [self.get_position(i, list(state.flatten())) for i in range(0, 16)]
135             X.append(chosen_piece)
136             return np.array([X])
137
138         def create_X(self, state, chosen_piece):
139             X = [self.get_position(i, list(state.flatten())) for i in range(0, 16)]
140             X.append(chosen_piece)
141             return np.array([X])
142
143         def train(self, replay_memory, batch_size):
144             '''Train the network'''
145             if len(replay_memory) < self.min_replay_size:
146                 return
147
148             # print('TRAINING')
149             batch_size = 64 * 2
150             minibatch = random.sample(replay_memory, batch_size)
151             # state + chosen_piece for you -> action (contains chosen_piece for next
152             #   player)
152             current_states = np.array([self.abbellire(state, chosen_piece)
153                                     for state, chosen_piece, action, reward,
154                                     ↪  new_current_state, done in minibatch])
154             current_qs = self.model.predict(current_states.reshape(batch_size, 17))
155             # new current state + chosen_piece for next player -> action (contains
156             #   chosen_piece for next player)
156             new_current_states = np.array([self.abbellire(new_current_state,
157             ↪  action[2])
157                                         for state, chosen_piece, action, reward,
158                                         ↪  new_current_state, done in
159                                         ↪  minibatch])
```

```
158        future_qs = self.target_model.predict(
159            new_current_states.reshape(batch_size, 17), verbose=0)
160        # exclude invalid moves from calculation
161        X = []
162        Y = []
163        for index, (current_state, chosen_piece, action, reward,
        ↪ new_current_state, done) in enumerate(minibatch):
164            if not done:
165                # max_future_q = np.max(future_qs[index])
166                # new_q = reward + self.gamma * max_future_q
167                max_future_q = reward + self.gamma * np.max(future_qs[index])
168            else:
169                # max_future_q = reward
170                max_future_q = reward
171
172            # 0 2 5
173            # 0 + 2 * 4 + 5 * 16 = 85
174            current_qs[index][action[0] + action[1] * 4 + action[2] * 16] = (
175                1 - self.lr) * current_qs[index][action[0] + action[1] * 4 +
                ↪ action[2] * 16] + self.lr * max_future_q
176
177            X.append(self.abbellire(current_state, chosen_piece))
178            Y.append(current_qs[index])
179
180        X = np.array(X).reshape(batch_size, 17)
181        Y = np.array(Y).reshape(batch_size, 4 * 4 * 16)
182        logging.debug(X)
183        logging.debug(Y)
184        self.model.fit(X, Y, batch_size=batch_size,
185                       verbose=1, shuffle=True, epochs=1)
186
187    def choose_piece(self, state: Any, piece_chosen_for_you: int):
188        '''Choose piece for the next guy to play'''
189        self.env.game.set_board(state)
190        pred = self.make_prediction(state, piece_chosen_for_you)
191        pred = self.nan_out_invalid_actions(-100, pred)
192        best_action = np.nanargmax(pred)
193        best_action = self.get_all_actions()[best_action]
194        return best_action[2]
195
196    def place_piece(self, state: Any, piece_chosen_for_you: int):
197        '''Choose position to move piece to based on the current state'''
198        self.env.game.set_board(state)
199        pred = self.make_prediction(state, piece_chosen_for_you)
200        pred = self.nan_out_invalid_actions(piece_chosen_for_you, pred)
201        best_action = np.nanargmax(pred)
202        best_action = self.get_all_actions()[best_action]
203        # print(f'Best action for place piece: {best_action}')
204        return best_action[0], best_action[1]
205
```

```python
206    def nan_out_invalid_actions(self, current_piece, prediction):
207        '''Zero out invalid moves'''
208        # zero out invalid moves
209        all_actions = self.get_all_actions()
210        for i in range(len(prediction)):
211            action = all_actions[i]
212            # print(action)
213            # print(current_piece)
214            if not self.env.game.check_if_move_valid(current_piece, action[0],
                ↪  action[1], action[2]):
215                prediction[i] = np.nan
216
217        return prediction
218
219    def run(self):
220        '''Run training of agent for x episodes'''
221        # ensure both model and target model have same set of weights at the
                ↪  start
222        self.target_model.set_weights(self.model.get_weights())
223
224        replay_memory = deque(maxlen=5000)
225        state = self.env.reset()
226        # number of episodes to train for
227        num_episodes = 2000
228
229        steps_to_update_target_model = 0
230
231        for episode in range(num_episodes):
232            if episode % 100 == 0:
233                self.model.save(f'/Volumes/USB/qn_weights.h5')
234
235            total_training_reward = 0
236            print(f'Episode: {episode}')
237            state = self.env.reset()
238            done = False
239            # initialise chosen piece with a random piece
240            # in reality, the opponent will choose a piece for you
241            chosen_piece = random.randint(0, 15)
242            while not done:
243                steps_to_update_target_model += 1
244
245                if random.random() < self.epsilon:
246                    action = self.env.action_space.sample()
247                    while not self.env.game.check_if_move_valid(chosen_piece,
                        ↪  action[0], action[1], action[2]):
248                        action = self.env.action_space.sample()
249                else:
250                    prediction = self.make_prediction(state, chosen_piece)
251                    prediction = self.nan_out_invalid_actions(
252                        chosen_piece, prediction)
```

```python
                        if np.all(np.isnan(prediction)):
                            action = self.env.action_space.sample()
                            while not self.env.game.check_if_move_valid(chosen_piece,
                            ↪  action[0], action[1], action[2]):
                                action = self.env.action_space.sample()
                        else:
                            action = np.nanargmax(prediction)
                            # get action at index of action
                            action = self.get_all_actions()[action]

                    new_state, reward, done, _ = self.env.step(
                        action, chosen_piece)

                    replay_memory.append(
                        (state, chosen_piece, action, reward, new_state, done))

                    if done:
                        logging.debug('GAME OVER')

                    if steps_to_update_target_model % 4 == 0 or done:
                        self.train(replay_memory, 32)

                    state = new_state
                    total_training_reward += reward

                    if done:
                        total_training_reward += 1

                        if steps_to_update_target_model >= 100:
                            self.target_model.set_weights(self.model.get_weights())
                            steps_to_update_target_model = 0
                        break

                    chosen_piece = action[2]

            if episode % 10 == 0:
                    logging.info(f'Testing win rate after {episode} episodes')
                    test(self)

            self.lr = self.decay_lr(self.lr, 0.0001, episode)

        self.env.close()
        self.model.save('/Volumes/USB/qn_weights.h5')

def main():
    dq_wins = 0
    for round in range(100):
        game = Quarto()
        dqn_agent = DQNAgent(game=game)
        dqn_agent.model = load_model('/Volumes/USB/qn_weights.h5')
```

```
302        game.set_players((RandomPlayer(game), DQNAgent(game=game)))
303        winner = game.run()
304        if winner == 1:
305            print('DQ wins')
306            dq_wins += 1
307        else:
308            print('Random wins')
309    print(f'DQ wins: {dq_wins/100}')
310
311 main()
```

6　Conclusion and Final Considerations

While working on the final project, I understood that complex algorithms do not necessarily outperform their simpler counterparts. I had spent a lot of time working on the Deep Q Network, and it didn't perform as well as expected. Despite hours of training, when the search space is too large, the algorithm takes an unreasonable amount of time to converge.

In spite of implementing several board symmetries based on the theory behind Quarto, I could not implement piece symmetries or board canonisation, which I'm sure would have reduced the search space.