

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.

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 polynomial 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

```
1 def naive_greedy(N):
2     goal = set(range(N))
3     covered = set()
4     solution = list()
5     all_lists = sorted(problem(N, seed=42), key=lambda l: len(l))
6     while goal != covered:
7         x = all_lists.pop(0)
8         if not set(x) < covered:
9             solution.append(x)
10            covered |= set(x)
11
12    print(
13        f"Naive greedy solution for N={N}: w={sum(len(_) for _ in solution)}
14        ↪ (bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
15    )
```

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.

```
1 def set_covering_problem_greedy(N, subsets, costs):
2     cost = 0
3     visited_nodes = 0
4     already_discovered = set()
5     final_solution = []
6     expected_solution = set(list(itertools.chain(*subsets)))
7     covered = set()
8     while covered != expected_solution:
9         subset = min(subsets, key=lambda s: costs[subsets.index(s)] /
10             ↪ (len(set(s)-covered) + 1))
11         final_solution.append(subset)
12         cost += costs[subsets.index(subset)]
13         visited_nodes = visited_nodes+1
14         covered |= set(subset)
15     print("NUMBER OF VISITED NODES: ", visited_nodes)
16     print("w: ", sum(len(_) for _ in final_solution))
17     print(
18         f"Naive greedy solution for N={N}: w={sum(len(_) for _ in final_solution)}
19         ↪ (bloat={(sum(len(_) for _ in final_solution)-N)/N*100:.0f}%)"
20     )
21     print(
22         f"My solution for N={N}: w={sum(len(_) for _ in final_solution)}
23         ↪ (bloat={(sum(len(_) for _ in final_solution)-N)/N*100:.0f}%)"
24     )
25     return final_solution, cost
26
27 for n in [5, 10, 50, 100, 500, 1000]:
28     subsets = problem(n, seed=SEED)
29     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.

```

1  from typing import Callable
2  from helpers import State, PriorityQueue
3  import numpy as np
4
5  class AStarSearch:
6      def __init__(self, N, seed=42):
7          # N is the number of elements to expect
8          self.N = N
9          self.seed = seed
10
11     def add_to_state(self, st, subset):
12         '''
13         Unnecessary function to add a subset to a state because we are using
14 ↳ the State class instead of a normal np.array
15         '''
16         state_list = st.copy_data().tolist()
17         state_list.append(subset)
18         return State(np.asarray(state_list, dtype=object))
19
20     def are_we_done(self, state):
21         '''
22         Check if we have reached the goal state (such that all elements are
23 ↳ covered in range(N))
24         '''
25         flattened_list = self.flatten_list(state.copy_data().tolist())
26         for i in range(self.N):
27             if i not in flattened_list:
28                 return False
29             # print("We are done")
30         return True
31
32     def flatten_list(self, l):
33         '''
34         Utility function to flatten a list of lists using itertools
35         '''
36         return list(itertools.chain.from_iterable(l))
37
38     def h(self, state):

```

```

37     '''
38     Heuristic Function h(n) = number of undiscovered elements
39     '''
40     num_undiscovered_elements = len(set(range(self.N)) -
41     ↪ set(self.flatten_list(state.copy_data().tolist())))
42     return num_undiscovered_elements
43
44 def astar_search(
45     self,
46     initial_state: State,
47     subsets: list,
48     parents: dict,
49     cost_of_each_state: dict,
50     priority_function: Callable,
51     unit_cost: Callable,
52 ):
53     frontier = PriorityQueue()
54     parents.clear()
55     cost_of_each_state.clear()
56
57     visited_nodes = 1
58     state = initial_state
59     parents[state] = None
60     cost_of_each_state[state] = 0
61     # to find length at the end without needed to flatten the state
62     discovered_elements = []
63
64     while state is not None and not self.are_we_done(state):
65         for subset in subsets:
66             # if this list has already been collected, skip
67             if subset in state.copy_data():
68                 # print("Already in")
69                 continue
70             new_state = self.add_to_state(state, subset)
71             state_cost = unit_cost(subset)
72             # if new_state not in cost_of_each_state or
73             ↪ cost_of_each_state[new_state] > cost_of_each_state[state] +
74             ↪ state_cost:
75             if new_state not in cost_of_each_state and new_state not in
76             ↪ frontier:
77                 parents[new_state] = state
78                 cost_of_each_state[new_state] = cost_of_each_state[state] +
79                 ↪ state_cost
80                 frontier.push(new_state, p=priority_function(new_state))
81             elif new_state in frontier and cost_of_each_state[new_state] >
82             ↪ cost_of_each_state[state] + state_cost:
83                 parents[new_state] = state
84                 cost_of_each_state[new_state] = cost_of_each_state[state] +
85                 ↪ state_cost
86             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
94     ↪ {len(cost_of_each_state):,} states")
95     print(f"Visited {visited_nodes} nodes")
96     print(
97         f"My solution for N={self.N}: w={sum(len(_) for _ in path[0])}
98         ↪ (bloat={(sum(len(_) for _ in
99         ↪ path[0]) - self.N) / self.N * 100:.0f}%)")
100
101     return list(reversed(path))
102
103 def search(self, constant_cost=False):
104     GOAL = State(np.array(range(self.N)))
105     subsets = problem(self.N, seed=self.seed)
106     initial_state = State(np.array([subsets[0]]))
107
108     parents = dict()
109     cost_of_each_state = dict()
110
111     self.astar_search(
112         initial_state = initial_state,
113         subsets = subsets,
114         parents = parents,
115         cost_of_each_state = cost_of_each_state,
116         priority_function = lambda state: cost_of_each_state[state] +
117         ↪ self.h(state),
118         unit_cost = lambda subset: 1 if constant_cost else len(subset)
119     )

```

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 N s.

Given $A = [2, 4, 5]$, $B = [2, 3, 1]$ and $C = [1, 2]$,

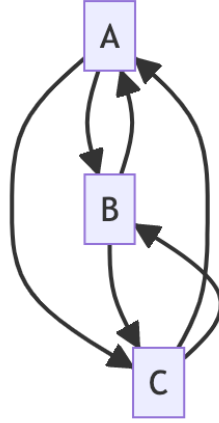


Figure 1: Fully connected graph

The heuristic function is slightly different:

$$h_i = \text{len}(s_i) - \text{len}(s_i \cap S_e)$$

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.

```

1  class AStarSearchFullyConnectedGraph:
2      def __init__(self, adjacency_list, list_values, N):
3          self.adjacency_list = adjacency_list
4          self.list_values = list_values
5          H = {}
6          for key in list_values:
7              # heuristic value is length of list
8              H[key] = len(list_values[key])
9          self.H = H
10         # holds the lists of each visited node
11         self.final_list = []
12         # N is the count of elements that should be in the final list
13         self.N = N
14         self.discovered_elements = set()
15

```

```

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 =
37         ↪ self.get_number_of_elements_not_in_second_list(self.list_values[n],
38         ↪ self.discovered_elements)
39     # if self.list_values[n] in self.final_list:
40     #     return 1000
41     return num_new_elements
42     # return self.H[n] / (num_new_elements + 1)
43
44 def get_node_with_least_h(self):
45     min_h = float("inf")
46     min_node = None
47     for node in self.adjacency_list:
48         if self.h(node) < min_h:
49             min_h = self.h(node)
50             min_node = node
51     return min_node
52
53 def get_node_with_least_h_and_not_in_final_list(self):
54     min_h = float("inf")
55     min_node = None
56     for node in self.adjacency_list:
57         if self.h(node) < min_h and node not in self.final_list:
58             min_h = self.h(node)
59             min_node = node
60     return min_node
61
62 # visited_node = [1, 2, 3]
63 # final_list = [[4, 5], [1]]
64 def are_we_done(self):
65     # flattened_list = list(itertools.chain.from_iterable(self.final_list))

```

```

64         for i in range(self.N):
65             if i not in self.discovered_elements:
66                 return False
67         print("We are done")
68         return True
69
70     def insert_unique_element_into_list(self, _list, element):
71         if element not in _list:
72             _list.append(element)
73         return _list
74
75     def a_star_algorithm(self):
76         # start_node is node with lowest cost
77         start_node = self.get_node_with_least_h()
78
79         open_list = [start_node]
80         closed_list = []
81
82         g = {}
83
84         g[start_node] = 0
85
86         parents = {}
87         parents[start_node] = start_node
88
89         while len(open_list) > 0:
90             n = None
91
92             # find a node with the highest value of f() - evaluation function
93             for v in open_list:
94                 if n == None or g[v] + self.h(v) > g[n] + self.h(n):
95                     n = v;
96
97             if n == None:
98                 print('Path does not exist!')
99                 return None
100
101             print(f"Visiting node: {n}")
102             self.final_list.append(self.list_values[n])
103             # self.discovered_elements.union(self.list_values[n])
104             # add list_values[n] to discovered_elements
105             for i in self.list_values[n]:
106                 self.discovered_elements.add(i)
107             print(len(self.discovered_elements))
108
109             # if the current node is the stop_node
110             # then we begin reconstructin the path from it to the start_node
111             if self.are_we_done():
112                 reconst_path = []
113

```

```

114         while parents[n] != n:
115             reconst_path.append(n)
116             n = parents[n]
117
118         reconst_path.append(start_node)
119
120         reconst_path.reverse()
121
122         print(f"Number of elements in final list:
123         ↪ {len(self.flatten_list(self.final_list))}")
124         print('Path found: {}'.format(reconst_path))
125         print(
126             f"My solution for N={N}: w={sum(len(_) for _ in
127             ↪ self.final_list)} (bloat={(sum(len(_) for _ in
128             ↪ self.final_list)-N)/N*100:.0f}%)"
129         )
130         return reconst_path
131
132     # for all neighbors of the current node do
133     for (m, weight) in self.get_neighbors(n):
134         values = self.list_values[m]
135         if m not in open_list and m not in closed_list:
136             # open_list.add(m)
137             open_list = self.insert_unique_element_into_list(open_list,
138             ↪ m)
139             # sort open_list by self.h
140             open_list = sorted(open_list, key=self.h)
141             parents[m] = n
142             g[m] = g[n] + weight
143
144         else:
145             if g[m] + self.h(m) > g[n] + self.h(n) + weight:
146                 g[m] = g[n] + weight
147                 parents[m] = n
148
149             # if m in closed_list:
150             #     closed_list.remove(m)
151             #     # open_list.add(m)
152             #     open_list =
153             ↪ self.insert_unique_element_into_list(open_list, m)
154             #     open_list = sorted(open_list, key=self.h)
155
156     open_list.remove(n)
157     open_list = sorted(open_list, key=self.h)
158     closed_list = self.insert_unique_element_into_list(closed_list, n)
159
160     print('Path does not exist!')
161     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 N s, 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 N s, 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

```
1 import random
2 import logging
3 logging.getLogger().setLevel(logging.INFO)
4
5 def custom_search(N, seed):
6     goal = set(range(N))
7     covered = set()
8     solution = list()
9     all_lists = problem(N, seed=42)
10    random.seed(seed)
11    random.shuffle(all_lists) #shuffle list to pop random
12    while goal != covered: #while set of covered nums is not equal to goal
13        x = all_lists.pop(0) #pick a list from all_lists
14        if not set(x) < covered: #if set of picked list is not a subset of
15            → covered
16            solution.append(x) #append it to the solution
17            covered |= set(x) #covered gets updated and becomes a union of
18            → covered plus picked set
19
20    logging.info(
21        f"custom search solution for N={N}: w={sum(len(_) for _ in solution)}
22        → (bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
23    )
24    logging.getLogger().setLevel(logging.DEBUG)
25    for N in [5, 10, 20, 100, 500, 1000]:
26        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 Dijkstra 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

```
1  import enum
2  from itertools import count
3  import logging
4  import random
5  from gx_utils import *
6  from heapq import heappush
7  from typing import Callable
8  import statistics
9  # import queues
10
11  logging.basicConfig(format="%(message)s", level=logging.INFO)
12
13  N = 1000
14  NUMBERS = {x for x in range(N)}
15
16
17  def problem(N, seed=None):
18      random.seed(seed)
19      return [
20          list(set(random.randint(0, N - 1) for n in range(random.randint(N //
21          ↪ 5, N // 2))))
22          for n in range(random.randint(N, N * 5))
23      ]
24
25  class State:
26      def __init__(self, list_numbers:set):
27          self.lists_ = list_numbers.copy()
28      def add(self,item):
29          self.lists_.add(item)
30          return self
31      def __hash__(self):
32          #return hash(bytes(self.lists_))
33          return hash(str(self.lists_))
34      def __eq__(self, other):
35          #return bytes(self.lists_) == bytes(other.lists_)
36          return str(self.lists_) == str(other.lists_)
37      def __lt__(self, other):
38          #return bytes(self.lists_) < bytes(other.lists_)
39          return str(self.lists_) < str(other.lists_)
40      def __str__(self):
41          return str(self.lists_)
```



```

41     def __repr__(self):
42         return repr(self.lists_)
43     def copy_data(self):
44         return self.lists_.copy()
45     def get_weight(self,ref_lists):
46         return len([x for n in self.lists_ for x in ref_lists[n]])
47     def get_items(self,ref_lists):
48         return set([x for n in self.lists_ for x in ref_lists[n]])
49
50
51     def goal_test(current_state:State,ref_lists):
52         """get all the members of the lists in the current_state and check if it
53         ↪ covers N"""
54
55         current_numbers = {x for n in current_state.lists_ for x in ref_lists[n]}
56         return current_numbers == NUMBERS
57
58     def valid_actions(current_state:State,ref_lists):
59         """returns set of indexes not currently added to this state"""
60         return {indx for indx,_ in enumerate(ref_lists) if indx not in
61         ↪ current_state.lists_}
62
63     def result(current_state,action):
64         next_state=State(current_state.copy_data()).add(action)
65         return next_state
66
67     def search(initial_state:State, ref_lists,priority_function:Callable):
68         frontier = PriorityQueue()
69         state = initial_state
70         state_count = 0
71         while state is not None and not goal_test(state,ref_lists):
72             for a in valid_actions(state,ref_lists):
73                 new_state = result(state,a)
74                 if new_state not in frontier:
75                     frontier.push(new_state,p=priority_function(new_state))
76                 elif new_state in frontier:
77                     pass
78             if frontier:
79                 state = frontier.pop()
80                 state_count+=1
81             else:
82                 state = None
83
84         logging.info(f"Found a solution with cost: {state.get_weight(ref_lists)}
85         ↪ and {state_count} number of visited states, last state: {state}")
86
87     def heuristic(state:State,ref_lists,N):
88         remained = NUMBERS - state.get_items(ref_lists)
89         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:
96         ↪ state.get_weight(ref_lists))
97
98         # #Depth_first
99         # search(initial_state, ref_lists,priority_function=lambda state:
100         ↪ -state.get_weight(ref_lists))
101
102         # #Heuristic
103         search(initial_state, ref_lists,priority_function=lambda state:
104         ↪ 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:

$$\text{len}(\text{distinct_elements}) \tag{1}$$

$$\text{len}(\text{distinct_elements})/(\text{num_duplicates} + 1) \tag{2}$$

$$\text{len}(\text{distinct_elements})/(\text{num_duplicates}+1)-\text{num_undiscovered_elements} \tag{3}$$

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

Table 1: Results of the algorithm

$$\text{len}(\text{distinct_elements})/(\text{num_undiscovered_elements} + 1) \quad (4)$$

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

3.2 Results

The following are the results of the algorithm after 1000 generations (only the best results are reported):

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.

```

1 def choose_mutation_rate(fitness_log):
2     # choose mutation rate based on change in fitness_log
3     if len(fitness_log) == 0:
4         return 0.2
5     if len(fitness_log) < 3:
6         considered_elements = len(fitness_log)
7     else:
8         considered_elements = 3
9     growth_rate = np.mean(np.diff(fitness_log[-considered_elements:]))
10    if growth_rate <= 0:
11        return 0.4
12    elif growth_rate < 0.5:
13        return 0.3
14    elif growth_rate < 1:
15        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

```
1 def scramble_mutation(genome):
2     '''
3     Randomly scrambles the genome.
4     '''
5     # select start and end indices to scramble
6     modified_genome = genome.copy()
7     start = random.randint(0, len(modified_genome) - 1)
8     end = random.randint(start, len(modified_genome) - 1)
9     # scramble the elements
10    modified_genome[start:end] = random.sample(modified_genome[start:end],
11        ↪ len(modified_genome[start:end]))
12    return return_best_genome(modified_genome, genome)
```

3.3.3 Swap Mutation

```
1 def swap_mutation(genome):
2     '''
3     Randomly swaps two elements in the genome.
4     '''
5     modified_genome = genome.copy()
6     index1 = random.randint(0, len(modified_genome) - 1)
7     index2 = random.randint(0, len(modified_genome) - 1)
8     modified_genome[index1], modified_genome[index2] = modified_genome[index2],
9     ↪ modified_genome[index1]
10    return return_best_genome(modified_genome, genome)
```

3.3.4 Inversion Mutation

```
1 def inversion_mutation(genome):
2     '''
3     Randomly inverts the genome.
4     '''
5     modified_genome = genome.copy()
6     # select start and end indices to invert
7     start = random.randint(0, len(modified_genome) - 1)
8     end = random.randint(start, len(modified_genome) - 1)
9     # invert the elements
10    modified_genome = modified_genome[:start] + modified_genome[start:end][::-1] +
11    ↪ modified_genome[end:]
12    return return_best_genome(modified_genome, genome)
```

3.4 Full Code

```
1 import numpy as np
2 import itertools
3
4 def calculate_fitness(genome):
5     '''
6     Calculates the fitness of the given genome.
7     The fitness is the number of unique elements
8     The weight is the total number of elements in the genome
9     '''
10    # fitness is number of distinct elements in genome
11    all_elements = []
12    distinct_elements = set()
13    weight = 0
14    for subset, gene in zip(prob, genome):
15        # if the particular element should be taken
16        if gene == 1:
17            distinct_elements.update(subset)
18            weight += len(subset)
19            all_elements += subset
20    num_duplicates = len(all_elements) - len(set(all_elements))
21    num_undiscovered_elements = len(set(range(N)) - distinct_elements)
22    # print(set(range(N)) - distinct_elements)
23    # print("num_undiscovered_elements", num_undiscovered_elements)
24    # return num_undiscovered_elements, -weight
25    # return len(distinct_elements), -weight
26    # return num_undiscovered_elements / (len(distinct_elements) + 1), -weight
27    return len(distinct_elements) / (num_undiscovered_elements + 1), -weight
28    # other potential fitness functions:
29    # return len(distinct_elements) / (num_duplicates + 1)
30    # return len(distinct_elements) / (num_duplicates + 1) -
31    ↪ num_undiscovered_elements, -weight
32    # return len(distinct_elements) / (num_undiscovered_elements + 1), -weight
33
34 def generate_element():
35     '''
36     Randomly generates offspring made up of 0s and 1s.
37     1 means the element is taken, 0 means it is not.
38     '''
39    genome = [random.randint(0, 1) for _ in range(N)]
40    fitness = calculate_fitness(genome)
41    # genome = np.random.choice([True, False], size=PROBLEM_SIZE)
42    return Individual(genome, fitness)
43
44 initial_population = [generate_element() for _ in range(POPULATION_SIZE)]
45
46 len(initial_population)
```

```

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,
84         ↪ num_generations))
85
86 def flip_mutation(genome, mutate_only_one_element=False):
87     '''
88     Flips random bit(s) in the genome.
89     Parameters:
90     mutate_only_one_element: If True, only one bit is flipped.
91     '''
92     modified_genome = genome.copy()
93     if mutate_only_one_element:
94         # flip a random bit
95         index = random.randint(0, len(modified_genome) - 1)
96         modified_genome[index] = 1 - modified_genome[index]

```



```

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

```

```

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],
153     ↪ 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],
163     ↪ 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]
175     ↪ + 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.

```

```

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)

```

```

238     else:
239         participant = max(random.sample(population, k=2), key=lambda x:
            ↪ x.fitness)
240         participant = Individual(participant.genome, participant.fitness)
241     return participant
242
243 def generate(population, generation):
244     '''
245     Create offspring from the population using either:
246     1. Cross Over + Mutation
247     2. Mutation
248     '''
249     # can either cross over between two parents or mutate a single parent
250     if random.random() < 0.2:
251         parent = tournament(population)
252         # if random.random() <= 0.3:
253         #     genome = mutation(parent.genome)
254         genome = mutation(parent.genome)
255         child = Individual(parent, calculate_fitness(parent))
256     else:
257         # crossover
258         parent1 = tournament(population)
259         parent2 = tournament(population)
260         genome = crossover(parent1.genome, parent2.genome)
261         # if random.random() <= 0.3:
262         #     genome = mutation(genome)
263         genome = mutation(genome)
264         child = Individual(genome, calculate_fitness(genome))
265
266     fitness_log.append((generation + 1, child.fitness[0]))
267
268     return child
269
270     best = max(initial_population, key=lambda x: x.fitness)
271
272     best_individual = max(initial_population, key=lambda x: x.fitness)
273     for i in range(NUM_GENERATIONS):
274         # create offspring
275         offspring = [generate(initial_population, i) for i in
            ↪ range(OFFSPRING_SIZE)]
276         # calculate fitness
277         # offspring = [Individual(child.genome, calculate_fitness(child.genome))
            ↪ for child in offspring]
278
279         initial_population = initial_population + offspring
280         initial_population = sorted(initial_population, key=lambda x: x.fitness,
            ↪ reverse=True)[:POPULATION_SIZE]
281
282         fittest_offspring = max(initial_population, key=lambda x: x.fitness)
283

```

```
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()'` otherwise 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.

```
1 for i in range(NUM_GENERATIONS):
2     # create offspring
3     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

```
1 # Should be used to init solution space, return a list of list
2 def select_rand_solution(full_input):
3     population = []
4     random.seed(None)
5     for i in range(POPULATION_SIZE):
6         population.append(random.sample(full_input, random.randint(1,
7                               ↪ len(full_input))))
8     return population
9
10 # check if one solution is valid
11 def goal_check(curr):
12     curr = [item for sublist in curr for item in sublist]
13     return set(curr) == set(range(N))
```

```

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,
43 ↪ mutate with 5% chance
44 def crossover(first_parent, second_parent):
45     slice_index_one = random.randint(0, min(len(first_parent[0]) - 1,
46     ↪ len(second_parent[0]) - 1))
47     child = first_parent[0][:slice_index_one] +
48     ↪ second_parent[0][slice_index_one:]
49     return child
50
51
52 # mutate child and return
53 def mutate_child(individual, problem_space):
54     index = random.randint(0, len(individual) - 1)
55     random_list = problem_space[random.randint(0, len(problem_space) - 1)]
56     random_gene = random_list[random.randint(0, len(random_list) - 1)]
57     individual = individual[:index] + individual[index+1:] + [random_gene]
58     return individual
59
60
61 def update_population(population, new_children):
62     new_population = population + new_children
63     sorted_population = sorted(new_population, key=lambda i: i[1], reverse=True)

```



```

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     ↪ sort
74     for individual in population:
75         current_individuals.append((individual, calculate_fitness(individual)))
76
77     current_individuals = sorted(current_individuals, key=lambda l: l[1],
78     ↪ reverse=True)
79
80     counter = 0
81     while counter < NR_OF_GENERATIONS:
82         # a) Select individuals with a good fitness score for reproduction.
83         cross_over_list = []
84         for i in range(OFFSPRING_SIZE):
85             parent_one = select_parents(current_individuals)
86             parent_two = select_parents(current_individuals)
87
88             # b) Let them produce offspring. Mutate with 5% chance
89             tmp_child = crossover(parent_one, parent_two)
90             if random.random() > 0.95:
91                 tmp_child = mutate_child(tmp_child, population)
92
93             cross_over_list.append((tmp_child, calculate_fitness(tmp_child)))
94
95     current_individuals = update_population(current_individuals,
96     ↪ cross_over_list)
97     counter += 1
98
99     for solution in current_individuals:
100         if goal_check(solution[0]):
101             logging.info(f'Best solution for N={N} was
102             ↪ {current_individuals[0][0]} \nWith a weight of {sum(len(_) for _
103             ↪ in current_individuals[0][0])}')
104         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:

```
1 def mutation(genome):  
2     '''  
3     Function mutates genome using .... strategy, etc.  
4     args:  
5     genome: str - Input genome  
6     '''
```

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:

```
1 def fitness_function(entry, goal_set):  
2     duplicates = len(entry) - len(set(tuple(entry)))  
3     miss = len(goal_set.difference(set(entry)))  
4     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:

```
1 return miss-duplicates
```

```
1 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

```
1  # helping functions
2
3  def lists_to_set(genome):
4      """
5          convert genome to set
6          :param genome: the sub-lists with random integers between 0 and N-1
7          :return: set of contained elements in the genome
8          """
9      list_elems = [single_elem for l in genome for single_elem in l]
10     s = set(list_elems)
11     return s
12
13 # find out how many duplicates there are in the population
14 def count_duplicates(genome):
15     """
16         Count how many duplicates there are in the genome
17         :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
22         ↪ groupby(sorted(list_elems))])
23     return duplicates
24 # to initialize the population
25 def create_population(STATE_SPACE, GOAL):
26     """
27     Initialize the population.
28     :param STATE_SPACE: List of lists generated from problem-function
29     :param GOAL: set of integers from 0 to N-1
30     :return: a list of tuples: (genome,fitness), for each individual in the
31     ↪ population.
32     """
33     population = []
34     for _ in range(POPULATION_SIZE):
35         individual = []
36         for _ in range(random.randint(1, len(STATE_SPACE))):
37             l = random.choice(STATE_SPACE)
38             if l not in individual: #check duplicates here
39                 individual.append(l)
40             #individual =
41             ↪ random.choices(STATE_SPACE,k=random.randint(1, len(STATE_SPACE)))
42             fitness = compute_fitness(individual, GOAL)
43             population.append((individual, fitness))
44     return population
45
46 def compute_fitness(genome, GOAL):
47     """
48     fitness is a tuple of (-#of_elems_missing, -#duplicates) which should be
49     ↪ maximized
50     :param genome: the sub-lists with random integers between 0 and N-1
51     :param GOAL: set of integers from 0 to N-1
52     :return: the fitness
53     """
54     # violated constraints, i.e. how many elements are missing
55     vc = GOAL.difference(lists_to_set(genome))
56     duplicates = count_duplicates(genome)
57     # it is worse to lack elements than having duplicates
58     fitness = (-len(vc), -duplicates)
59     return fitness
60
61 def goal_check(genome, GOAL):
62     """
63     Check if all required elements are in the genome
64     :param genome: the sub-lists with random integers between 0 and N-1
65     :param GOAL: set of integers from 0 to N-1
66     :return: boolean value if goal reached or not
67     """

```

```

64     return GOAL==lists_to_set(genome)
65
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
78     ↪ (arithmetic sum)
79     n_slots = POPULATION_SIZE*(POPULATION_SIZE+1)/2
80     wheel_number = random.randint(1,n_slots)
81     curr_parent = 0
82     parent_number = POPULATION_SIZE
83     increment = POPULATION_SIZE-1
84     while wheel_number > parent_number:
85         curr_parent +=1
86         parent_number +=increment
87         increment -= 1
88     return ranked_population[curr_parent]
89
90 # make one child from each cross-over, and mutate with low prob
91 def cross_over(parent1, parent2, STATE_SPACE, mutation_prob = 0.1):
92     """
93     Compute cross-over between two selected parents. Mutate child with
94     ↪ mutation_prob.
95     :param parent1: individual
96     :param parent2: individual
97     :param STATE_SPACE: List of lists generated from problem-function
98     :param mutation_prob: the probability to perform mutation
99     :return: the child created
100     """
101     cut1 = random.randint(0,len(parent1[0]))
102     cut2 = random.randint(0,len(parent2[0]))
103     child = parent1[0][:cut1]+parent2[0][cut2:]
104     if random.random() < mutation_prob:
105         mutate(child, STATE_SPACE)
106     return child
107
108 def mutate(child, STATE_SPACE):
109     """
110     Replace one list in the child with a random one from the state space.
111     :param child:
112     :param STATE_SPACE:

```

```

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

```

```

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),
170                 ↪ parent_selection(population)
171             child = cross_over(parent1,parent2, STATE_SPACE, mutation_prob)
172             child_fitness = compute_fitness(child, GOAL)
173             offspring.append((child,child_fitness))
174             population = update_population_plus(population, offspring)
175             #population = update_population_comma(offspring)
176             best_curr = sorted(population, key=lambda l:l[1], reverse=True)[0]
177             mutation_prob, mutation_param = update_mutation_prob(best_sol, best_curr,
178                 ↪ mutation_param, i)
179             if goal_check(best_curr[0],GOAL) and best_curr[1] > best_sol[1]:
180                 best_sol = best_curr
181                 found_in_iter = i
182             logging.info(f'Best solution found in {found_in_iter} iters and has weight
183                 ↪ {-best_sol[1][1]}')
184         return best_sol
185
186 # main
187
188 # settings
189 POPULATION_SIZE = 50
190 OFFSPRING_SIZE = 30
191 ITTERS = 100
192
193 for N in [5,10,20,50,100,1000,2000]:
194     best_sol = solve_problem(N)
195     print(f'N = {N}')
196     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.

```
1      # it is worse to lack elements than having duplicates
2      fitness = (-len(vc), -duplicates)
3      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.

```
1  def cross_over(parent1, parent2, STATE_SPACE):
2      cut1 = random.randint(0, len(parent1[0]))
3      cut2 = random.randint(0, len(parent2[0]))
4      child = parent1[0][:cut1] + parent2[0][cut2:]
5      # dynamic_threshold = do some computation here to derive probability from the
6      ↪ change in fitness
7      # if random.random() < dynamic_threshold
7          mutate(child, STATE_SPACE)
8      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


```

1 from itertools import compress
2 from collections import namedtuple
3 N = 5
4 POPULATION_SIZE = 10
5 OFFSPRING_SIZE = 2
6 GENERATIONS = 5
7 PROB = 0.5 # probability to choose 1 for each one of the locus in the
  ↳ population
8 Individual = namedtuple('Individual', ('genome', 'fitness', 'goal_reached',
  ↳ 'w'))
9 # this function evaluates the fitness and if the goal was reached
10 def fitness_goal_eval(list_of_lists, genome, goal):
11     current_goal = goal
12     solution = list(compress(list_of_lists, genome))
13     # fitness = 0
14     new_elements = 0
15     repeated_elements = 0
16     w = 0
17     goal_reached = False
18
19     if len(solution) == 0:
20         return 0, False, 0
21
22     for list_ in solution:
23         list_length = len(list_)
24         list_ = set(list_)
25         cg_length = len(current_goal)
26         current_goal = current_goal - list_
27         cg_new_length = len(current_goal)
28
29         # fitness += cg_length - cg_new_length # new elements (positive)
30         # fitness += (cg_length - cg_new_length) - list_length # repeated
  ↳ elements (negative)
31         new_elements += cg_length - cg_new_length # new elements
32         repeated_elements += list_length - (cg_length - cg_new_length) #
  ↳ repeated elements
33
34         w += list_length
35
36     if cg_new_length == 0:
37         goal_reached = True
38
39     fitness = new_elements - repeated_elements
40
41     return fitness, goal_reached, w
42
43
44 def generate_population(list_of_lists, goal):
45     population = list()
46

```

```

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

```

```

91     for g in range(GENERATIONS):
92         population = sorted(population, key=lambda i: i.fitness,
93                               ↪ reverse=True)[:POPULATION_SIZE-OFFSPRING_SIZE]
94
95         for i in range(OFFSPRING_SIZE):
96             p1 = select_parent(population,
97                               ↪ tournament_size=int(0.2*genome_size))
98             p2 = select_parent(population,
99                               ↪ tournament_size=int(0.2*genome_size))
100             o = cross_over(p1, p2, genome_size, list_of_lists, goal)
101             fitness, goal_reached, w = fitness_goal_eval(list_of_lists, o,
102                                                           ↪ goal)
103             o = mutation(o, genome_size, k=2)
104
105             population.append(Individual(o, fitness, goal_reached, w))
106
107     for i in population:
108         if i.goal_reached:
109             return i, population
110
111     print(f"No solution for current population (N={N})")
112     return None, population
113
114 N = 500
115 POPULATION_SIZE = 100
116 OFFSPRING_SIZE = 20
117 GENERATIONS = 200
118 PROB = 0.5
119
120 logging.getLogger().setLevel(logging.INFO)
121
122 solution, population = genetic_algorithm()
123 if solution != None:
124     logging.info(
125         f" Genetic algorithm solution for N={N:,}: "
126         + f"fitness={solution.fitness:,} "
127         + f"w={solution.w:,} "
128         + f"(bloat={solution.w/N*100:.0f}%) "
129     )
130     INFO:root: Genetic algorithm solution for N=500: fitness=-1,980 w=2,980
131     ↪ (bloat=596%)
132     POPULATION_SIZE = 50
133     OFFSPRING_SIZE = 20
134     GENERATIONS = 200
135     PROB = 0.5
136
137     logging.getLogger().setLevel(logging.INFO)
138
139     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),
6             ↪ k=2*cut_size))
7         ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
8     else:
9         cut_size = int(genome_size*0.2)
10        new_genome_cut = tuple(random.choices([1, 0], weights=(1, 99),
11            ↪ k=2*cut_size))
12        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.

```
1 if random.random() < threshold or some_fitness_based_condition:
2     # crossover
3 elif random.random() < threshold:
4     # crossover + mutate
5 elif ....:
6     # 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

```
1 import random
2 import logging
3 import numpy as np
4 from collections import namedtuple
5 def problem(N, seed=None):
6     random.seed(seed)
7     return [
8         list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5,
9             ↪ N // 2))))
10        for n in range(random.randint(N, N * 5))
11    ]
12 def tournament(population, tournament_size=2):
13     return max(random.choices(population, k=tournament_size), key=lambda i:
14         ↪ i.fitness)
15
16 def w(genome):
17     return sum(len(_) for _ in genome)
18
19 def covering(genome):
20     s = set()
21     for _ in genome:
22         s = s.union(set(_))
23     return len(s)
24
25 def intersection(lst1, lst2):
```

```

24     lst3 = [value for value in lst1 if value in lst2]
25     return lst3
26
27 def shuffle(g1,g2,g3):
28     a = [l for l in g1 if l not in g3]
29     b = [l for l in g2 if l not in g3]
30     gnew = g3.copy()
31
32     if a:
33         c = 1
34     else:
35         c = 0
36     for i in range(max(len(a),len(b))):
37         if c :
38             if a and i < len(a):
39                 gnew.append(a[i])
40             if b:
41                 c = 0
42
43         else:
44             if b and i < len(b):
45                 gnew.append(b[i])
46             if a:
47                 c = 1
48
49     return gnew
50
51 def cross_over(g1, g2):
52     g3 = intersection(g1,g2)
53     g3 = shuffle(g1,g2,g3)
54     return g3
55
56
57 def mutation(genome):
58
59     mutation = random.choice(all_lists)
60     if mutation in genome:
61         genome.remove(mutation)
62     else:
63         genome.append(mutation)
64
65     return genome
66
67 def create_population(mu):
68     population = []
69     for i in range(mu):
70         g = []
71         while covering(g) != N:
72             if len(g) < N*2:
73                 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,
102                        ↪ reverse=True)[:mu]
103
104 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)

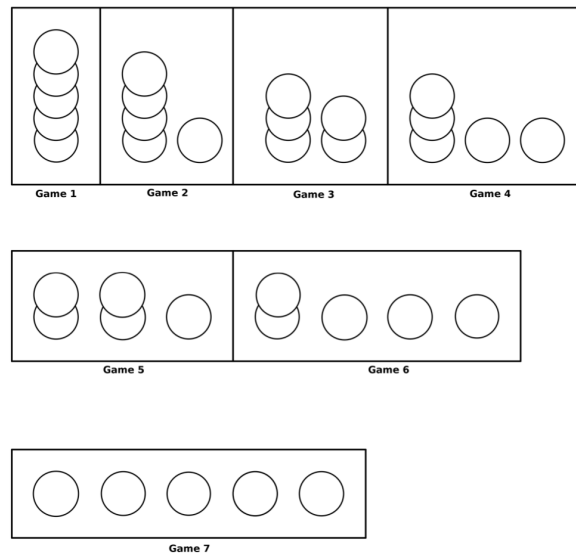
4.1 Solution

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-Elves 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. \square

Figure 2: Fixed Rules

```

1  from collections import Counter
2  from copy import deepcopy
3  from itertools import accumulate
4  import logging
5  from operator import xor
6  import random
7  from typing import Callable
8
9  from lib import Genome, Nim, Nimply
10
11
12  class FixedRuleNim:
13      def __init__(self):
14          self.num_moves = 0
15          self.OFFSPRING_SIZE = 30
16          self.POPULATION_SIZE = 100
17          self.GENERATIONS = 100
18          self.nim_size = 5
19
20      def nim_sum(self, nim: Nim):
21          '''
22          Returns the nim sum of the current game board
23          by taking an XOR of all the rows.
24          Ideally, agent should try to leave nim sum of 0 at the end of turn
25          '''
26          *__, result = accumulate(nim.rows, xor)
27          return result

```

```

28
29     def init_population(self, population_size, nim: Nim):
30         '''
31         Initialize population of genomes,
32         key is rule, value is number of sticks to take
33         The rules currently are:
34         1. If one pile, take  $x$  number of sticks from the pile.
35         2. If two piles:
36             a. If 1 pile has 1 stick, wipe out the pile
37             b. If 2 piles have multiple sticks, take  $x$  sticks from any pile
38         3. If three piles and two piles have the same size, remove all sticks
39         ↪ from the smallest pile
40         4. If  $n$  piles and  $n-1$  piles have the same size, remove  $x$  sticks from
41         ↪ the smallest pile until it is the same size as the other piles
42         '''
43         population = []
44         for i in range(population_size):
45             # rules 3 and 4 are fixed (apply for 3 or more piles)
46             # different strategies for different rules (situations on the
47             ↪ board)
48             individual = {
49                 'rule_1': [0, random.randint(0, (nim.num_rows - 1) * 2)],
50                 'rule_2a': [random.randint(0, 1), random.randint(0,
51                 ↪ (nim.num_rows - 1) * 2)],
52                 'rule_2b': [random.randint(0, 1), random.randint(0,
53                 ↪ (nim.num_rows - 1) * 2)],
54                 'rule_3': [nim.rows.index(min(nim.rows)), min(nim.rows)],
55                 'rule_4': [nim.rows.index(max(nim.rows)), max(nim.rows) -
56                 ↪ min(nim.rows)]
57             }
58             genome = Genome(individual)
59             population.append(genome)
60         return population
61
62     def statistics(self, nim: Nim):
63         '''
64         Similar to Squillero's cooked function to get possible moves
65         and statistics on Nim board
66         '''
67         # logging.info('In statistics')
68         # logging.info(nim.rows)
69         stats = {
70             'possible_moves': [(r, o) for r, c in enumerate(nim.rows) for o
71             ↪ in range(1, c + 1) if nim.k is None or o <= nim.k],
72             # 'possible_moves': [(row, num_objects) for row in
73             ↪ range(nim.num_rows) for num_objects in range(1,
74             ↪ nim.rows[row]+1)],
75             'num_active_rows': sum(o > 0 for o in nim.rows),
76             'shortest_row': min((x for x in enumerate(nim.rows) if x[1] > 0),
77             ↪ key=lambda y: y[1])[0],

```

```

68         'longest_row': max((x for x in enumerate(nim.rows)), key=lambda
        ↪ y: y[1])[0],
69         # only 1-stick row and not all rows having only 1 stick
70         '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]),
71         'nim_sum': self.nim_sum(nim)
72     }
73
74     brute_force = []
75     for move in stats['possible_moves']:
76         tmp = deepcopy(nim)
77         tmp.nimming_remove(*move)
78         brute_force.append((move, self.nim_sum(tmp)))
79     stats['brute_force'] = brute_force
80
81     return stats
82
83     def strategy(self):
84         '''
85         Returns the best move to make based on the statistics
86         '''
87     def engine(nim: Nim):
88         stats = self.statistics(nim)
89         if stats['num_active_rows'] == 1:
90             # logging.info('m1')
91             return Nimply(stats['shortest_row'], random.randint(1,
92             ↪ stats['possible_moves'][0][1]))
93         elif stats["num_active_rows"] % 2 == 0:
94             # logging.info('m2')
95             if max(nim.rows) == 1:
96                 return Nimply(stats['longest_row'], 1)
97             else:
98                 pile = random.choice([i for i, x in enumerate(nim.rows)
99                 ↪ if x > 1])
100                 return Nimply(pile, nim.rows[pile] - 1)
101         elif stats['num_active_rows'] == 3:
102             # logging.info('m3')
103             unique_elements = set(nim.rows)
104             # check if 2 rows have the same number of sticks
105             two_rows_with_same_elements = False
106             for element in unique_elements:
107                 if nim.rows.count(element) == 2:
108                     two_rows_with_same_elements = True
109                     break
110
111             if len(nim.rows) == 3 and two_rows_with_same_elements:
112                 # remove 1 stick from the longest row
113                 logging.info(nim.rows)
114                 return Nimply(stats['longest_row'], max(max(nim.rows) -
115                 ↪ nim.rows[stats['shortest_row']], 1))

```

```

113         else:
114             # do something random
115             return Nimply(*random.choice(stats['possible_moves']))
116     elif stats['num_active_rows'] >= 4:
117         # logging.info('m4')
118         counter = Counter()
119         for element in nim.rows:
120             counter[element] += 1
121         if len(counter) == 2:
122             if counter.most_common()[0][1] == 1:
123                 # remove x sticks from the smallest pile until it is
124                 # → the same size as the other piles
125                 return Nimply(stats['shortest_row'],
126                               # → max(nim.rows[stats['shortest_row']] -
127                               # → counter.most_common()[1][0], 1))
128                 return random.choice(stats['possible_moves'])
129         else:
130             # logging.info('m5')
131             return random.choice(stats['possible_moves'])
132     return engine
133
134 def random_agent(self, nim: Nim):
135     """
136     Random agent that takes a random move
137     """
138     stats = self.statistics(nim)
139     return random.choice(stats['possible_moves'])
140
141 def battle(self, opponent, num_games=1000):
142     """
143     Battle this agent against another agent
144     """
145     wins = 0
146     for _ in range(num_games):
147         nim = Nim()
148         while not nim.goal():
149             nim.nimming_remove(*self.play(nim))
150             if sum(nim.rows) == 0:
151                 break
152             nim.nimming_remove(*opponent.play(nim))
153             if sum(nim.rows) == 0:
154                 wins += 1
155     return wins
156
157 if __name__ == '__main__':
158     rounds = 20
159     evolved_agent_wins = 0
160     for i in range(rounds):
161         nim = Nim(5)
162         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
31                               ↪ range(1, c+1)]
32         move_found = False
33         for move in all_possible_moves:
34             replicated_nim = deepcopy(nim)
35             replicated_nim.nimming_remove(*move)
36             if self.nim_sum(replicated_nim) == 0:
37                 nim.nimming_remove(*move)
38                 move_found = True
39                 break
40         # if a valid move not found, return random move
41         if not move_found:
42             move = random.choice(all_possible_moves)
43             nim.nimming_remove(*move)
44
45         # logging.info(f"Move {self.num_moves}: Removed {move[1]} objects from
46         ↪ row {move[0]}")
47         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.

```

1      '''
2      In this file, I will try to implement Nim where there is an evolved set of
3      ↪ rules/strategies.
4      For each scenario, I will have a set of rules that will be used to determine the
5      ↪ best move.
6      They are obtained from discussion with friends and from the paper "The Game of
7      ↪ Nim" by Ryan Julian
8      The rules currently are:
9      1. If one pile, take  $x$  number of sticks from the pile.
10     2. If two piles:
11         a. If 1 pile has 1 stick, take  $x$  sticks
12         b. If 2 piles have multiple sticks, take  $x$  sticks from the larger pile
13     3. If three piles and two piles have the same size, remove all sticks from the
14     ↪ smallest pile
15     4. If  $n$  piles and  $n-1$  piles have the same size, remove  $x$  sticks from the smallest
16     ↪ pile until it is the same size as the other piles
17     '''
18
19     from collections import Counter, namedtuple
20     from copy import deepcopy
21     from itertools import accumulate
22     import logging
23     from operator import xor
24     import random
25     from typing import Callable
26
27     from lib import Genome, Nim, Nimply
28
29     class BrilliantEvolvedAgent:
30         def __init__(self):
31             self.num_moves = 0
32             self.OFFSPRING_SIZE = 200
33             self.POPULATION_SIZE = 50
34             self.GENERATIONS = 100
35             self.nim_size = 5
36
37         def nim_sum(self, nim: Nim):
38             '''
39             Returns the nim sum of the current game board

```



```

35         by taking an XOR of all the rows.
36         Ideally, agent should try to leave nim sum of 0 at the end of turn
37         '''
38         *_ , result = accumulate(nim.rows, xor)
39         return result
40
41     def init_population(self, population_size, nim: Nim):
42         '''
43         Initialize population of genomes,
44         key is rule, value is number of sticks to take
45         The rules currently are:
46         1. If one pile, take  $x$  number of sticks from the pile.
47         2. If two piles:
48             a. If 1 pile has 1 stick, wipe out the pile
49             b. If 2 piles have multiple sticks, take  $x$  sticks from any pile
50         3. If three piles and two piles have the same size, remove all sticks
51         ↪ from the smallest pile
52         4. If  $n$  piles and  $n-1$  piles have the same size, remove  $x$  sticks from the
53         ↪ smallest pile until it is the same size as the other piles
54         5. If none of the above rules apply, just pick a random pile and take a
55         ↪ random number of sticks
56         '''
57         population = []
58         for i in range(population_size):
59             # rules 3 and 4 are fixed (apply for 3 or more piles)
60             # different strategies for different rules (situations on the board)
61             individual = {
62                 'rule_1': [0, random.randint(0, (self.nim_size - 1) * 2)],
63                 'rule_2a': [random.randint(0, 1), random.randint(0,
64                 ↪ (self.nim_size - 1) * 2)],
65                 'rule_2b': [random.randint(0, 1), random.randint(0,
66                 ↪ (self.nim_size - 1) * 2)],
67                 'rule_3': [nim.rows.index(min(nim.rows)), min(nim.rows)],
68                 'rule_4': [nim.rows.index(max(nim.rows)), max(nim.rows) -
69                 ↪ min(nim.rows)]
70             }
71             genome = Genome(individual)
72             population.append(genome)
73         return population
74
75     def crossover(self, parent1, parent2, crossover_rate):
76         '''
77         Crossover function to combine two parents into a child
78         '''
79         child = {}
80         for rule in parent1.rules:
81             if random.random() < crossover_rate:
82                 child[rule] = parent1.rules[rule]
83             else:
84                 child[rule] = parent2.rules[rule]

```

```

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],

```

```

120     # 'possible_moves': [(row, num_objects) for row in
    ↪ range(nim.num_rows) for num_objects in range(1,
    ↪ nim.rows[row]+1)],
121     'num_active_rows': sum(o > 0 for o in nim.rows),
122     'shortest_row': min((x for x in enumerate(nim.rows) if x[1] > 0),
    ↪ key=lambda y: y[1])[0],
123     'longest_row': max((x for x in enumerate(nim.rows)), key=lambda y:
    ↪ y[1])[0],
124     # only 1-stick row and not all rows having only 1 stick
125     '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]),
126     'nim_sum': self.nim_sum(nim)
127 }
128
129 brute_force = []
130 for move in stats['possible_moves']:
131     tmp = deepcopy(nim)
132     tmp.nimming_remove(*move)
133     brute_force.append((move, self.nim_sum(tmp)))
134 stats['brute_force'] = brute_force
135
136 return stats
137
138 def strategy(self, genome: dict):
139     '''
140     Returns the best move to make based on the statistics
141     '''
142     def evolution(nim: Nim):
143         stats = self.statistics(nim)
144         if stats['num_active_rows'] == 1:
145             num_to_leave = genome.rules['rule_1'][1]
146             # see which move will leave the most sticks
147             most_destructive_move = max(stats['possible_moves'], key=lambda
    ↪ x: x[1])
148             if num_to_leave >= most_destructive_move[1]:
149                 # remove only 1 stick
150                 return Nimply(most_destructive_move[0], 1)
151             else:
152                 # make the move that leaves the desired number of sticks
153                 move = [(row, num_objects) for row, num_objects in
    ↪ stats['possible_moves'] if nim.rows[row] - num_objects ==
    ↪ num_to_leave]
154                 if len(move) > 0:
155                     return Nimply(*move[0])
156                 else:
157                     # make random move
158                     return Nimply(*random.choice(stats['possible_moves']))
159
160         elif stats['num_active_rows'] == 2:
161             # rule 2a

```

```

162     if stats['1_stick_row']:
163         # if there is a 1-stick row, have to choose between wiping it
164         ↪ out or taking from the other row
165     if genome.rules['rule_2a'][0] == 0:
166         # wipe out the 1-stick row
167         logging.info('wiping out 1-stick row')
168         pile = [row for row in range(nim.num_rows) if
169         ↪ nim.rows[row] == 1][0]
170         return Nimply(pile, 1)
171     else:
172         # take out the desired number of sticks from the other
173         ↪ row
174         pile = random.choice([index for index, x in
175         ↪ enumerate(nim.rows) if x > 1])
176         num_objects_to_remove = max(1, nim.rows[pile] -
177         ↪ genome.rules['rule_2a'][1])
178         # move = [(row, num_objects) for row, num_objects in
179         ↪ stats['possible_moves'] if nim.rows[row] -
180         ↪ num_objects == genome.rules['rule_2a'][1]]
181         return Nimply(pile, num_objects_to_remove)
182 # rule 2b
183 # both piles have many elements, take from either the smallest or
184 ↪ the largest pile
185 else:
186     if genome.rules['rule_2b'][0] == 0:
187         # take from the smallest pile
188         pile = stats['shortest_row']
189         num_objects_to_remove = max(1, nim.rows[pile] -
190         ↪ genome.rules['rule_2b'][1])
191         return Nimply(pile, num_objects_to_remove)
192     else:
193         # take from the largest pile
194         pile = stats['longest_row']
195         num_objects_to_remove = max(1, nim.rows[pile] -
196         ↪ genome.rules['rule_2b'][1])
197         return Nimply(pile, num_objects_to_remove)
198
199 elif stats['num_active_rows'] == 3:
200     unique_elements = set(nim.rows)
201     # check if 2 rows have the same number of sticks
202     two_rows_with_same_elements = False
203     for element in unique_elements:
204         if nim.rows.count(element) == 2:
205             two_rows_with_same_elements = True
206             break
207
208 if len(nim.rows) == 3 and two_rows_with_same_elements:
209     # remove 1 stick from the longest row
210     return Nimply(stats['longest_row'], max(max(nim.rows) -
211     ↪ nim.rows[stats['shortest_row']], 1))

```

```

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
211             ↪ size as the other piles
212             return Nimply(stats['shortest_row'],
213                             ↪ max(nim.rows[stats['shortest_row']] -
214                             ↪ counter.most_common()[1][0], 1))
215         # else:
216         #     return random.choice(stats['possible_moves'])
217
218     # for large number of piles, general rule to remove all but 1 stick
219     ↪ from a random pile
220     if stats["num_active_rows"] % 2 == 0:
221         if nim.rows[stats['longest_row']] == 1:
222             return Nimply(stats['longest_row'], 1)
223         else:
224             pile = random.choice([i for i, x in enumerate(nim.rows) if x
225                                 ↪ > 1])
226             return Nimply(pile, nim.rows[pile] - 1)
227
228     else:
229         # this is a fixed rule, does not have random component
230         # rule from the paper Ryan Julian: The Game of Nim
231         # If n piles and n-1 piles have the same size, remove x sticks
232         ↪ from the smallest pile until it is the same size as the other
233         ↪ piles
234         # check if only 1 pile has a different number of sticks
235         # just make a random move if all else fails
236         return random.choice(stats['possible_moves'])
237     return evolution
238
239 def random_agent(self, nim: Nim):
240     """
241     Random agent that takes a random move
242     """
243     stats = self.statistics(nim)
244     return random.choice(stats['possible_moves'])
245
246 def dumb_agent(self, nim: Nim):
247     """
248     Agent that takes one element from the longest row
249     """
250     stats = self.statistics(nim)

```

```

244         return (stats['longest_row'], 1)
245
246     def aggressive_agent(self, nim: Nim):
247         '''
248         Agent that takes the largest possible move
249         '''
250         stats = self.statistics(nim)
251         if stats['num_active_rows'] % 2 == 0:
252             return random.choice(stats['possible_moves'])
253         else:
254             row = stats['longest_row']
255             return (row, nim.rows[row])
256
257         # stats = self.statistics(nim)
258         # return max(stats['possible_moves'], key=lambda x: x[1])
259
260     def calculate_fitness(self, genome):
261         '''
262         Calculate fitness by playing the genome's strategy against a random
263         ↪ agent
264         (cannot use nim sum agent as it is too good)
265         '''
266         wins = 0
267         for i in range(5):
268             nim = Nim(5)
269             player = 0
270             engine = self.strategy(genome)
271             while not nim.goal():
272                 if player == 0:
273                     move = engine(nim)
274                     nim.nimming_remove(*move)
275                     player = 1
276                 else:
277                     nim.nimming_remove(*self.random_agent(nim))
278                     player = 0
279             winner = 1 - player
280             if winner == 0:
281                 wins += 1
282         return wins / 5
283
284     def select_survivors(self, population: list, num_survivors: int):
285         '''
286         Select the best genomes from the population
287         '''
288         return sorted(population, key=lambda x: x.fitness,
289             ↪ reverse=True)[:num_survivors]
290
291     def learn(self, population_size=100, mutation_rate=0.1, crossover_rate=0.7,
292         ↪ nim: Nim = None):
293         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,
304             ↪ 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
2     ↪ on Probabilities)
3     from itertools import accumulate
4     from operator import xor
5     import random
6     import numpy as np

```



```

7     from lib import Nim
8
9     class EvolvedAgent1:
10         '''
11         Plays Nim using a set of rules that are evolved
12         '''
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.
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 play_nim(self, nim: Nim, prob_list: list):
26             '''
27             GA can choose between the following strategies:
28             1. Randomly pick any row and any number of elements from that row
29             2. Pick the shortest row
30             3. Pick the longest row
31             4. Pick based on the nim-sum of the current game board
32             '''
33             all_possible_moves = [(r, o) for r, c in enumerate(nim.rows) for o in
34                                   ↪ range(1, c+1)]
35             strategies = {
36                 'nim_sum': random.choice([move for move in all_possible_moves if
37                                           ↪ self.nim_sum(deepcopy(nim).nimming_remove(*move)) == 0]),
38                 'random': random.choice(all_possible_moves),
39                 'all_elements_shortest_row': (nim.rows.index(min(nim.rows)),
40                                               ↪ min(nim.rows)),
41                 '1_element_shortest_row': (nim.rows.index(min(nim.rows)), 1),
42                 'random_element_shortest_row': (nim.rows.index(min(nim.rows)),
43                                                 ↪ random.randint(1, min(nim.rows))),
44                 'all_elements_longest_row': (nim.rows.index(max(nim.rows)),
45                                              ↪ max(nim.rows)),
46                 '1_element_longest_row': (nim.rows.index(max(nim.rows)), 1),
47                 'random_element_longest_row': (nim.rows.index(max(nim.rows)),
48                                                ↪ random.randint(1, max(nim.rows))),
49             }
50
51             p = random.random()
52             strategy = None
53             if p < prob_list[0]:
54                 strategy = strategies['random']
55             elif p >= prob_list[0] and p < prob_list[1]:

```

```

50         strategy =
51             ↪ random.choice([strategies['all_elements_shortest_row'],
52                             ↪ strategies['1_element_shortest_row'],
53                             ↪ strategies['random_element_shortest_row']])
54     elif p >= prob_list[1] and p < prob_list[2]:
55         strategy = random.choice([strategies['all_elements_longest_row'],
56                                   ↪ strategies['1_element_longest_row'],
57                                   ↪ strategies['random_element_longest_row']])
58     else:
59         strategy = strategies['nim_sum']
60
61     nim.nimming_remove(*strategy)
62     self.num_moves += 1
63     return sum(nim.rows)
64
65 def play(self, nim: Nim):
66     '''
67     Play the game of Nim using the evolved rules
68     '''
69     prob_list = [0.25, 0.5, 0.75, 1]
70     prob_list = self.evolve_probabilities(nim, prob_list, 20, 5)
71     self.play_nim(nim, prob_list)
72
73 def crossover(self, p1, p2):
74     '''
75     Crossover between two parents
76     '''
77     return np.random.choice(p1 + p2, size=4, replace=True)
78
79 def evolve_probabilities(self, nim: Nim, prob_list: list,
80 ↪ num_generations: int, num_children: int):
81     '''
82     Evolve the probabilities of the strategies
83     '''
84     # create initial population
85     population = [prob_list for _ in range(num_children)]
86     # create initial fitness scores
87     fitness_scores = [self.play(nim, p) for p in population]
88     # create initial parents
89     parents = [population[i] for i in np.argsort(fitness_scores)[:2]]
90     # create new population
91     new_population = []
92     for _ in range(num_generations):
93         # create children
94         for _ in range(num_children):
95             p1 = random.choice(parents)
96             p2 = random.choice(parents)
97             child = self.crossover(p1, p2)
98             # child = []
99             # 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
102                    ↪ np.argsort(fitness_scores)[:2]]
103        # create new population
104        new_population = []
105        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.

```

```

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 =
36         ↪ -1, beta: int = 1, max_depth: int = 7):
37         '''
38         Depth-limited Minimax algorithm to find the best move with alpha-beta
39         ↪ pruning and depth limit
40         '''
41         logging.info("Depth ", depth)
42         if depth == 0 or nim.goal() or depth == max_depth:
43             # logging.info("Depth ", depth)
44             # logging.info("Nim goal ", nim.goal())
45             return self.evaluate(nim, maximizing_player)
46
47         if maximizing_player:
48             value = -math.inf
49             for r, c in enumerate(nim.rows):
50                 for o in range(1, c+1):
51                     # make copy of nim object before running a nimming operation
52                     replicated_nim = deepcopy(nim)
53                     replicated_nim.nimming_remove(r, o)
54                     value = max(value, self.minmax(replicated_nim, depth-1,
55                     ↪ False, alpha, beta))
56                     alpha = max(alpha, value)
57                     if beta <= alpha:
58                         logging.info("Pruned")
59                         break
60             return value
61         else:
62             value = math.inf
63             for r, c in enumerate(nim.rows):
64                 for o in range(1, c+1):
65                     # make copy of nim object before running a nimming operation
66                     replicated_nim = deepcopy(nim)
67                     replicated_nim.nimming_remove(r, o)
68                     value = min(value, self.minmax(replicated_nim, depth-1, True,
69                     ↪ 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,
83                 ↪ False)))
84             # sort possible moves by the value returned by minimax
85             possible_moves.sort(key=lambda x: x[2], reverse=True)
86             # return the best move
87             return possible_moves[0][0], possible_moves[0][1]
88
89 def battle(self, opponent, num_games=1000):
90     """
91     Battle this agent against another agent
92     """
93     wins = 0
94     for _ in range(num_games):
95         nim = Nim()
96         while not nim.goal():
97             nim.nimming_remove(*self.play(nim))
98             if sum(nim.rows) == 0:
99                 break
100             nim.nimming_remove(*opponent.play(nim))
101             if sum(nim.rows) == 0:
102                 wins += 1
103         return wins
104
105 if __name__ == "__main__":
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 representation 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.

```

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

```

```

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
91                     # ↪ {}".format(state.rows, (r, o), val))
92                     self.set_Q(hash_list(state.rows), (r, o), val)
93                 else:
94                     logging.info("State already registered: {} and action:
95                     ↪ {}".format(state.rows, (r, o)))

```



```

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

```

```

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

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

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.

```
1 logging.basicConfig(level=logging.INFO)
2
3 def hash_list(l):
4     '''
5     Hashes a list of integers into a string
6     '''
7     return "-".join([str(i) for i in l])
8
9
10 def unhash_list(l):
11     '''
12     Unhashes a string of integers into a list
13     '''
14     return [int(i) for i in l.split("-")]
15
16
17 def decay(value, decay_rate):
18     return value * decay_rate
19
20
21 class NimRLMonteCarloAgent:
22     def __init__(self, num_rows: int, epsilon: float = 0.3, alpha: float = 0.5,
23         ↪ gamma: float = 0.9):
24         """Initialize agent."""
25         self.num_rows = num_rows
26         self.epsilon = epsilon
27         self.alpha = alpha
28         self.gamma = gamma
29         self.current_state = None
30         self.previous_state = None
31         self.previous_action = None
32         self.G = dict()
33         self.state_history = []
34
35     def get_action(self, state: Nim):
36         """Return action based on epsilon-greedy policy."""
37         if random.random() < self.epsilon:
38             action = random.choice(self.get_possible_actions(state))
39             if (hash_list(state.rows), action) not in self.G:
40                 self.G[(hash_list(state.rows), action)] = random.uniform(1.0,
41                     ↪ 0.01)
42             return action
43         else:
44             max_G = -math.inf
45             best_action = None
```

```

44         for r, c in enumerate(state.rows):
45             for o in range(1, c+1):
46                 if (hash_list(state.rows), (r, o)) not in self.G:
47                     self.G[(hash_list(state.rows), (r, o))] =
48                         ↪ random.uniform(1.0, 0.01)
49                     G = self.G[(hash_list(state.rows), (r, o))]
50                 else:
51                     G = self.G[(hash_list(state.rows), (r, o))]
52                 if G > max_G:
53                     max_G = G
54                     best_action = (r, o)
55             return best_action
56
57 def update_state(self, state, reward):
58     self.state_history.append((state, reward))
59
60 def learn(self):
61     target = 0
62
63     for state, reward in reversed(self.state_history):
64         self.G[state] = self.G.get(state, 0) + self.alpha * (target -
65             ↪ self.G.get(state, 0))
66         target += reward
67
68     self.state_history = []
69     self.epsilon -= 10e-5
70
71 def compute_reward(self, state: Nim):
72     return 0 if state.goal() else -1
73
74 def get_possible_actions(self, state: Nim):
75     actions = []
76     for r, c in enumerate(state.rows):
77         for o in range(1, c+1):
78             actions.append((r, o))
79     return actions
80
81 def get_G(self, state: Nim, action: tuple):
82     return self.G.get((hash_list(state.rows), action), 0)
83
84 def battle(self, opponent, training=True):
85     player = 0
86     agent_wins = 0
87     for episode in range(rounds):
88         self.current_state = Nim(num_rows=self.num_rows)
89         while True:
90             if player == 0:
91                 action = self.get_action(self.current_state)
92                 self.current_state.nimming_remove(*action)
93                 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

I have discussed with Karl Wennerstrom and Diego Gasco.

My reinforcement agent initially performed very poorly until I realised that there was a bug in `update_Q`, where I forgot to hash the nim state before checking the presence of the compound key in the Q dictionary. Hence, it was reinitialised every time, effectively rendering random performance and wasting a big chunk of my time.

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!

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):

```
1      """
2      Agents based on different strategies playing Nim (description here:
↪      https://en.wikipedia.org/wiki/Nim)
3          1. Agent based on rules
4          2. Agent based on evolved rules
5          3. Agent using minmax
6          4. Agent using reinforcement learning
7
8      @Author: Karl Wennerström in collaboration with Erik Bengtsson (s306792)
9      """
10
11     # ...
12
13     # %% Q.2 Create own strategy based on cooked information
14
15     # strategy maker: play by the rules
16     def make_strategy(agent: Evolvable_agent) -> Callable:
17         def evolvable(state: Nim) -> Nimply:
18             data = cook_status(state)
19
20             # rule 1
21             if data['active_rows_number'] == 1:
22                 row, elem = agent.rule1(data)
23                 ply = Nimply(row, elem)
24
25             elif data['one_multiple_elem_row']: # all rows but one have a single
↪             elem
26
27             # rule2
28             if data['active_rows_number'] % 2 == 0: # even rows
29                 row, elem = agent.rule2(data)
30                 ply = Nimply(row, elem)
31
32             # rule 3
33             else: # odd rows
34                 row, elem = agent.rule3(data)
35                 ply = Nimply(row, elem)
36
37             elif not data['one_multiple_elem_row']: # multiple rows are with
↪             multiple elems (or also only ones)
38
39             # rule 4
40             if data['active_rows_number'] % 2 == 0:
41                 row, elem = agent.rule4(data)
```



```

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
73                 ↪ enumerate(pm) ]}')
74         else:
75             row = pm[int(index)][0]
76             elems = pm[int(index)][1]
77             break
78     return Nimply(row, elems)
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:
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
108         ↪ elem
109         if data['active_rows_number'] % 2 == 0:
110             move = [(r, e) for (r, e) in data["possible_moves"] if
111                 ↪ state.rows[r] - e == 1][0]
112             ply = Nimply(move[0], move[1])
113         else:
114             move = [(r, e) for (r, e) in data["possible_moves"] if
115                 ↪ state.rows[r] - e == 0 and r not in
116                 ↪ data['single_elem_rows_index']][0]
117             ply = Nimply(move[0], move[1])
118         else:
119             row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
120             num_objects = random.randint(1, state.rows[row])
121             ply = Nimply(row, num_objects)
122     return ply
123
124 # %% EVOLUTION STRATEGY DESCRIBED
125
126 """
127 (mu, lambda)-strategy
128 1. Create population with the same set of rules but different parameters
129 ↪ for each rule
130 2. k individuals competes in a tournament where the winner becomes a
131 ↪ parent
132 3. Perform cross_over between two parents and mutate (aggregate random
133 ↪ rule, e.g. mean(both parents' rule)) with certain prob
134 4. Generate offspring where OFFSPRING_SIZE >> POPULATION_SIZE
135 5. Fitness for offsprings corresponds to how many games are won against
136 ↪ their 'siblings'
137 6. Slice new population from fittest offspring
138 7. Repeat step 2-6 GENERATION times
139 """
140
141 # %% Evolution strategy-functions

```

```

134 def init_population():
135     """Initialize population"""
136     pop = []
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
144     ↪ against different opponents"""
145     for ind in individuals:
146         fitness = []
147         for idx, strat in enumerate(OPPONENTS):
148             wins = 0
149             for match in range(NUM_MATCHES):
150                 wins += head2head(ind, strat)
151             fitness.append(wins / NUM_MATCHES)
152             ind.fitness = tuple(fitness)
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,
174                 ↪ reverse=True)[:POPULATION_SIZE]
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,
181                     ↪ reverse=True)[0]

```

```

181         return best_contestor
182
183
184 def cross_over(parent1: Evolvable_agent, parent2: Evolvable_agent,
185 ↪ mutation_prob: float) -> Evolvable_agent:
186     """Generate new individual by cross-over of parents' rules"""
187     rules = [rule for rule in parent1.rules.keys()]
188     new_rules = {}
189     child = Evolvable_agent(NIM_SIZE)
190     for k in rules:
191         which_parent = random.randint(1, 2)
192         new_rules[k] = parent1.rules[k] if which_parent == 1 else
193 ↪ parent2.rules[k]
194     if random.random() < mutation_prob:
195         rule = random.choice(rules)
196         if rule == 'rule_1':
197             new_rules[rule] = random.randint(0, (NIM_SIZE - 1) * 2)
198         else:
199             new_rules[rule] = [random.randint(0, 1), random.randint(0,
200 ↪ (NIM_SIZE - 1) * 2)]
201     child.rules = new_rules
202     return child
203
204
205 def create_offspring(population: list, k: int, mutation_prob: float) -> list:
206     """Create new offspring"""
207     offspring = []
208     for _ in range(OFFSPRING_SIZE):
209         p1 = tournament(population=population, k=k)
210         p2 = tournament(population=population, k=k)
211         child = cross_over(parent1=p1, parent2=p2,
212 ↪ mutation_prob=mutation_prob)
213         offspring.append(child)
214     return offspring
215
216
217 def get_next_generation(offspring: list) -> list:
218     """Find the best individuals in the new generation"""
219     calc_fitness(offspring)
220     return fittest_individuals(offspring)
221
222
223 # %% PLAYING FUNCTIONS
224 def evaluate(strategy1: Callable, strategy2: Callable) -> float:
225     """Play two strategies against each other and evaluate their performance
226     ↪ """
227     players = (strategy1, strategy2)
228     won = 0
229
230     for m in range(EVAL_MATCHES):

```

```

226         nim = Nim(NIM_SIZE)
227         player = 0
228         while nim:
229             ply = players[player](nim)
230             nim.nimming(ply)
231             player = 1 - player
232         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 always 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 until 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')
282                 break
283         return bestVal
284     else:
285         bestVal = np.inf
286         new_state = deepcopy(state)
287         ply = optimal_strategy(new_state)
288         new_state.nimming(ply)
289         value = minmax(new_state, True, alpha, beta)
290         bestVal = min(bestVal, value)
291         return bestVal
292
293 def best_move(state: Nim):
294     data = cook_status(state)
295     for ply in data['possible_moves']:
296         tmp_state = deepcopy(state)
297         tmp_state.nimming(ply)
298         score = minmax(tmp_state, my_turn=False)
299         if score > 0:
300             break
301     return ply
302
303 # %% Q4 - RL
304
305 """
306 Reinforcement learning agent to play Nim
307
308 Idea:
309     Play using Upper Confidence Trees (UCT), a Monte Carlo Tree Search (MCTS)
310 → algorithm, popular when trade-off between
311     finding best-so-far and finding a better one
312
313 Need:
314     * All possible states (TODO: sort state so that e.g. 1 1 0 == 1 0 1)
315     * Init with value 0 and visits 0
316     * Actions for each state (based on data)
317     * Simulate function
318     * Reward function
319
320 Outline:
321     1. Selection (select an unvisited node) with highest UCT
322     2. Expand to that node
323     3. Simulate from that node until termination
324     4. Backpropagate and update node with statistics

```

```

324         *  $N(v)$  - number of visits for node  $v$ 
325         *  $Q(v)$  - value/reward playing from that node
326
327     UCT:
328          $uct(v_i, v) = Q(v_i)/N(v_i) + c\sqrt{\log(N(v))/N(v_i)}$ , which prefers
↪ child nodes with small  $N(v_i)$ 
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         ↪ -----
342         def __init__(self, nim_size: int, random_factor=0.2,
343                     exploration_factor=np.sqrt(2)): # explore with 20%, exploit
344             ↪ with 80%
345             self.nim_size = nim_size
346             self.current_state = None
347             self.previous_state = None
348             self.__init_states(nim_size)
349             self.random_factor = random_factor
350             self.c = exploration_factor
351
352         def __init_states(self, nim_size: int):
353             """find all possible board positions"""
354             states = {}
355             rows = [i * 2 + 1 for i in range(nim_size)]
356             elem_ranges = list(itertools.combinations([range(n + 1) for n in
357             ↪ rows], r=nim_size))
358             all_states = list(itertools.product(*elem_ranges[0]))
359
360             for state in all_states:
361                 states[state] = {}
362                 states[state]['visits'] = 0
363                 states[state]['value'] = 0
364                 states[state]['child_states'] = self.__init_child_states(state)
365             self.states = states
366             # last state is the initial board
367             self.current_state = all_states[-1]
368             self.states[self.current_state]['visits'] = 1
369
370         def __init_child_states(self, state):
371             """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
388         ↳ child nodes) """
389         visits_parent = self.states[self.current_state]['visits']
390         best_state = None
391         best_uct = -np.inf
392         for child_state in self.states[self.current_state]['child_states']:
393             visits_child = self.states[child_state]['visits']
394             wins_child = self.states[child_state]['value']
395             uct = wins_child / (visits_child + 1) + self.c *
396                 ↳ (np.log(visits_parent) / (visits_child + 1)) ** (1 / 2)
397             if uct > best_uct:
398                 best_uct = uct
399                 best_state = child_state
400         return best_state
401
402     def random_selection(self):
403         """Explore and move to random state"""
404         next_state =
405             ↳ random.choice(tuple(self.states[self.current_state]['child_states']))
406         return next_state
407
408     def expand(self, next_state):
409         """Expand to the found next state. Return the ply that takes agent
410         ↳ there"""
411         self.previous_state = self.current_state
412         self.current_state = next_state
413         ply = self.__next_ply()
414         return ply
415
416     def __next_ply(self):
417         """ Find ply that takes agent from previous state to current
418         ↳ 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]]
419                 ↪ - ply[1] == self.current_state[ply[0]][0]
420         return ply
421
422     def simulate(self, opponent: Callable, n_matches: int):
423         """Simulate game of nim vs opponent by letting RL agent play randomly
424         ↪ from current state"""
425         players = (opponent, pure_random) # rl agent is second since played
426         ↪ move to get here
427         nim = Nim(self.nim_size)
428         won = 0
429         for match in range(n_matches):
430             # forbidden stuff
431             nim._rows = list(self.current_state) # play from current state
432
433             player = 0
434             while nim:
435                 ply = players[player](nim)
436                 nim.nimming(ply)
437                 player = 1 - player
438             if player == 0:
439                 won += 1
440
441         # update results
442         self.backpropagate(n_matches, won)
443
444     def backpropagate(self, visits: int, reward: int):
445         """Update results after simulating `visits` times game from current
446         ↪ state"""
447         self.states[self.current_state]['visits'] += visits
448         self.states[self.current_state]['value'] += reward
449
450     # TRAINING
451     ↪ -----
452     def learn_to_play(self, opponents: list, n_sims: int, n_matches: int):
453         """Simulate the game from original state. For each move, simulate the
454         ↪ outcome n_matches times.
455         Keep moving until board is empty, then repeat n_sims times."""
456         for opponent in opponents:
457             for n in tqdm(range(n_sims), desc="Iterations, %s"
458                             ↪ %opponent.__name__):
459                 # always start from initial state in a new simulation
460                 nim = Nim(self.nim_size)
461                 self.current_state = nim.rows
462
463                 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
465         self.simulate(opponent, n_matches)
466
467     def get_statistics(self):
468         """Print overview of number of visits and wins for a visited
469         ↪ 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
473                 ↪ {state[1]}')
474
475     def policy(self, state: Nim) -> Nimply:
476         """The policy, i.e. the next move for the current state"""
477         self.current_state = state.rows
478         ns = self.selection()
479         ply = self.expand(next_state=ns)
480         return ply
481
482     # %% MAIN
483     import argparse
484
485     if __name__ == '__main__':
486
487         # VARIABLES
488         NIM_SIZE = 3
489         NUM_MATCHES = 100
490         EVAL_MATCHES = 100
491
492         # INPUT
493         parser = argparse.ArgumentParser()
494         parser.add_argument("-t", "--task", dest="task", default=1,
495                             help="Which task should run? Choose from 1, 2, 3 or
496                             ↪ 4.", type=int)
497
498         args = parser.parse_args()
499         print(f"Task: {args.task}")
500
501         # -----TASK 1 - PLAYING THE OPTIMAL STRATEGY
502         ↪ -----
503         if args.task == 1:
504             play_nim(optimal_strategy, optimal_strategy)
505             # 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

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. 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.

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

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

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:

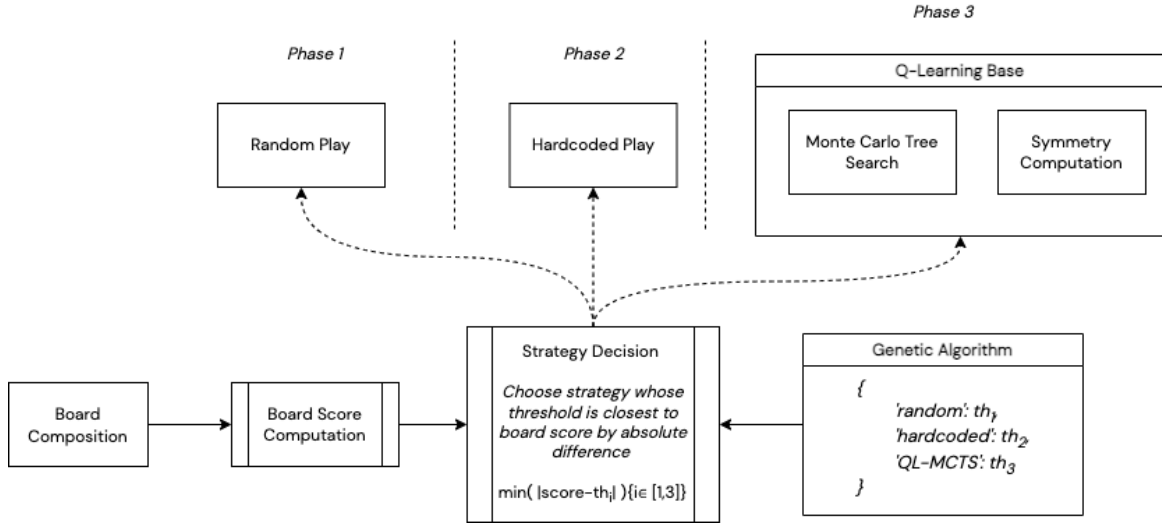


Figure 3: Hybrid Agent

- 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 ??, 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 ??.

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 Code of Hybrid Agent and Sub Players

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

5.3.1 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 intuition that the slowest but most powerful algorithm should be used last.

```
1  '''  
2  Genetic Algorithm for Quarto  
3  '''  
4  import sys  
5  sys.path.insert(0, '..')  
6  
7  import tqdm  
8  import random  
9  import logging  
10 import json  
11 import itertools  
12 from copy import deepcopy  
13 from lib.players import Player, RandomPlayer  
14 from quarto.objects import Quarto  
15 from lib.scoring import score_board
```

```

16 from QLMCTS import QLearningPlayer
17 from Hardcoded.hardcoded import HardcodedPlayer
18
19 logging.basicConfig(level=logging.DEBUG)
20
21 class Genome:
22     def __init__(self, thresholds, fitness):
23         self.thresholds = thresholds
24         self.fitness = fitness
25
26     def set_fitness(self, fitness):
27         self.fitness = fitness
28
29     def set_thresholds(self, thresholds):
30         self.thresholds = thresholds
31
32     def toJSON(self):
33         return {
34             'thresholds': self.thresholds,
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
85
86 def ensure_correct_ordering(self, new_thresholds):
87     if new_thresholds['random'] > new_thresholds['hardcoded']:
88         new_thresholds['random'], new_thresholds['hardcoded'] =
89             ↪ new_thresholds['hardcoded'], new_thresholds['random']
90     if new_thresholds['hardcoded'] > new_thresholds['ql-mcts']:
91         new_thresholds['hardcoded'], new_thresholds['ql-mcts'] =
92             ↪ new_thresholds['ql-mcts'], new_thresholds['hardcoded']
93     if new_thresholds['random'] > new_thresholds['hardcoded']:
94         new_thresholds['random'], new_thresholds['hardcoded'] =
95             ↪ new_thresholds['hardcoded'], new_thresholds['random']
96     return new_thresholds
97
98 def crossover(self, genome1, genome2):
99     new_thresholds = {}
100     for key in genome1.thresholds:
101         new_thresholds[key] = random.choice(
102             [genome1.thresholds[key], genome2.thresholds[key]])
103
104     # make sure that value for random < hardcoded < ql-mcts
105     new_thresholds = self.ensure_correct_ordering(new_thresholds)
106     return Genome(new_thresholds, 0)
107
108 def mutate(self, genome):
109     new_thresholds = {}
110     genome_thresholds = genome.thresholds
111     if random.random() < 0.4:
112         new_thresholds['random'] = random.random() * \
113             self.GENOME_VAL_UPPER_BOUND
114         new_thresholds['hardcoded'] = random.choice(
115             [genome_thresholds['random'], genome_thresholds['random'] +

```

```

112         random.random() * (self.GENOME_VAL_UPPER_BOUND -
113                               ↪ genome_thresholds['random'])))
114     new_thresholds['ql-mcts'] = random.choice(
115         [genome_thresholds['hardcoded'], genome_thresholds['hardcoded'] +
116          random.random() * (self.GENOME_VAL_UPPER_BOUND -
117                               ↪ genome_thresholds['hardcoded'])])
118
119     new_thresholds = self.ensure_correct_ordering(new_thresholds)
120
121     assert new_thresholds['random'] < new_thresholds['hardcoded'] <
122     ↪ new_thresholds['ql-mcts']
123
124     return Genome(new_thresholds, 0)
125 return genome
126
127 def evolve(self, num_generations=50):
128     self.population_size = 50
129     self.offspring_size = 10
130     population = self.generate_population(self.population_size)
131
132     pbar = tqdm.tqdm(total=num_generations)
133     for gen in range(num_generations):
134         pbar.update(1)
135         logging.debug('Generation: {}'.format(gen))
136         offspring = []
137         for i in range(self.offspring_size):
138             parent1 = random.choice(population)
139             parent2 = random.choice(population)
140             child = self.crossover(parent1, parent2)
141             child = self.mutate(child)
142             child.fitness = self.play_game(child.thresholds, num_games=5)
143             offspring.append(child)
144         population += offspring
145         population = sorted(
146             population, key=lambda x: x.fitness,
147             ↪ reverse=True)[:self.population_size]
148
149         if gen % 5 == 0:
150             logging.info('Saving population')
151             with open('/Volumes/USB/population3.json', 'w') as f:
152                 json.dump([genome.toJSON() for genome in population], f)
153
154         # return the best genome's thresholds
155         return population[0].thresholds
156
157 def play_game(self, thresholds, num_games=10):
158     wins = 0
159     for game in range(num_games):
160         logging.debug('Game: {}'.format(game))
161         state = Quarto()

```

```

158     player = 0
159
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
163
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)
169
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)
174
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
188                     ↪ self.current_state.check_if_move_valid(self.current_state.get_s
189                     ↪ action[0], action[1], next_piece) is False:
190                         action = self.random_player.place_piece()
191                         next_piece = self.random_player.choose_piece()
192                 self.current_state.select(
193                     self.current_state.get_selected_piece())
194                 self.current_state.place(action[0], action[1])
195                 self.current_state.set_selected_piece(next_piece)
196                 self.current_state.switch_player()
197                 player = 1 - player
198
199             elif key == 'hardcoded':
200                 # play using hardcoded strategy
201                 logging.debug('hardcoded')
202                 self.previous_state = deepcopy(self.current_state)
203                 winning_piece, position =
204                     ↪ self.hardcoded.hardcoded_strategy_get_move(
205                     ↪ self.current_state)
206                 next_piece = self.hardcoded.hardcoded_strategy_get_piece(
207                     self.current_state)

```

```

205         while
206             ↪ self.current_state.check_if_move_valid(self.current_state.get_s
207             ↪ position[0], position[1], next_piece) is False:
208                 winning_piece, position =
209                 ↪ self.hardcoded.hardcoded_strategy_get_move(
210                     self.current_state)
211                 next_piece =
212                 ↪ self.hardcoded.hardcoded_strategy_get_piece(
213                     self.current_state)
214                 self.current_state.select(state.get_selected_piece())
215                 self.current_state.place(position[0], position[1])
216                 self.current_state.set_selected_piece(next_piece)
217                 self.current_state.switch_player()
218                 player = 1 - player
219
220     else:
221         # play using QL-MCTS
222         logging.debug('ql-mcts')
223         self.ql_mcts.previous_state = deepcopy(
224             self.current_state)
225         action = self.ql_mcts.get_action(self.current_state)
226         self.ql_mcts.previous_action = action
227         self.ql_mcts.current_state.select(
228             self.current_state.get_selected_piece())
229         self.ql_mcts.current_state.place(action[0], action[1])
230         self.ql_mcts.current_state.set_selected_piece(
231             action[2])
232         self.ql_mcts.current_state.switch_player()
233         player = 1 - player
234
235     else:
236         # opponent is random
237         action = self.random_player.place_piece()
238         next_piece = self.random_player.choose_piece()
239         while
240             ↪ self.current_state.check_if_move_valid(self.current_state.get_selec
241             ↪ action[0], action[1], next_piece) is False:
242                 action = self.random_player.place_piece()
243                 next_piece = self.random_player.choose_piece()
244                 # WARNING: very often stuck in this loop
245         self.current_state.select(
246             self.current_state.get_selected_piece())
247         self.current_state.place(action[0], action[1])
248         self.current_state.set_selected_piece(next_piece)
249         self.current_state.switch_player()
250         player = 1 - player
251
252     if self.current_state.check_is_game_over():
253         if 1 - self.current_state.check_winner() == 0:
254             print("Agent wins")

```

```

249         wins += 1
250         # TODO: QL reward update
251     else:
252         print("Player 2 wins")
253         break
254
255     # fitness is the percentage of games won
256     logging.debug(f"Win rate: {wins/num_games}")
257     return wins/num_games
258
259 def choose_piece(self):
260     '''
261     Choose piece for next player to place
262     '''
263     thresholds = self.thresholds
264
265     # game is stored in parent
266     self.current_state = self.get_game()
267
268     board_score = score_board(self.current_state)
269
270     differences = [abs(board_score - thresholds[key])
271                   for key in thresholds]
272     min_diff = min(differences)
273     index = differences.index(min_diff)
274     key = list(thresholds.keys())[index]
275
276     if key == 'random':
277         logging.debug('random')
278         # play randomly
279         action = self.random_player.place_piece()
280         next_piece = self.random_player.choose_piece()
281         while
282             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_piec
283             ↪ action[0], action[1], next_piece) is False:
284                 action = self.random_player.place_piece()
285                 next_piece = self.random_player.choose_piece()
286         return next_piece
287
288     elif key == 'hardcoded':
289         # play using hardcoded strategy
290         logging.debug('hardcoded')
291         self.previous_state = deepcopy(self.current_state)
292         winning_piece, position = self.hardcoded_strategy_get_move(
293             self.current_state)
294         next_piece = self.hardcoded_strategy_get_piece(
295             self.current_state)
296         while
297             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_piec
298             ↪ position[0], position[1], next_piece) is False:

```



```

295         winning_piece, position = self.hardcoded_strategy_get_move(
296             self.current_state)
297         next_piece = self.hardcoded_strategy_get_piece(
298             self.current_state)
299         return next_piece
300
301     else:
302         # play using QL-MCTS
303         logging.debug('ql-mcts')
304         self.ql_mcts.previous_state = deepcopy(
305             self.current_state)
306         action = self.ql_mcts.get_action(self.current_state)
307         self.ql_mcts.previous_action = action
308         self.ql_mcts.current_state.select(
309             self.current_state.get_selected_piece())
310         return action[2]
311
312     def place_piece(self):
313         # python passes by reference
314         # agent will use the state, etc. to update the Q-table
315         # this function also wipes the MCTS tree
316         self.current_state = self.get_game()
317         thresholds = self.thresholds
318
319         self.ql_mcts.clear_and_set_current_state(self.current_state)
320
321         while True:
322             # board score is the number of couples and triplets on the board
323             # it is indicative of the change of the board state
324             board_score = score_board(self.current_state)
325
326             differences = [abs(board_score - thresholds[key])
327                            for key in thresholds]
328             min_diff = min(differences)
329             index = differences.index(min_diff)
330             key = list(thresholds.keys())[index]
331
332             if key == 'random':
333                 logging.debug('random')
334                 # play randomly
335                 action = self.random_player.place_piece()
336                 next_piece = self.random_player.choose_piece()
337                 while
338                     ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_
339                     ↪ action[0], action[1], next_piece) is False:
340                     action = self.random_player.place_piece()
341                     next_piece = self.random_player.choose_piece()
342                 return action[0], action[1]

```

```

343         # play using hardcoded strategy
344         logging.debug('hardcoded')
345         self.previous_state = deepcopy(self.current_state)
346         winning_piece, position = self.hardcoded_strategy_get_move(
347             self.current_state)
348         next_piece = self.hardcoded_strategy_get_piece(
349             self.current_state)
350         while
351             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_
352             ↪ position[0], position[1], next_piece) is False:
353                 winning_piece, position = self.hardcoded_strategy_get_move(
354                     self.current_state)
355                 next_piece = self.hardcoded_strategy_get_piece(
356                     self.current_state)
357         return position[0], position[1]
358
359     else:
360         # play using QL-MCTS
361         logging.debug('ql-mcts')
362         self.ql_mcts.previous_state = deepcopy(
363             self.current_state)
364         action = self.ql_mcts.get_action(self.current_state)
365         self.ql_mcts.previous_action = action
366         return action[0], action[1]
367
368     def test_thresholds(self):
369         thresholds = {'random': 1.090773081612301,
370                      'hardcoded': 5.790328881747581, 'ql-mcts':
371                      ↪ 9.251997327518943}
372         win_rate = self.play_game(thresholds, num_games=10)
373         print(win_rate)
374
375     if __name__ == "__main__":
376         final_player = FinalPlayer()
377         # best_thresholds = final_player.evolve()
378         # print(best_thresholds)
379
380         final_player.test_thresholds()

```

5.3.2 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):
21     def __init__(self, quarto: Quarto = None):
22         if quarto is None:
23             quarto = Quarto()
24         super().__init__(quarto)
25         self.BOARD_SIDE = 4
26
27     def check_if_winning_piece(self, state, piece):
28         '''
29         Simulate placing the piece on the board and check if the game is over
30         '''
31
32         for i in range(self.BOARD_SIDE):
33             for j in range(self.BOARD_SIDE):
34                 if state.check_if_move_valid(piece, i, j, -100):
35                     cloned_state = deepcopy(state)
36                     cloned_state.select(piece)
37                     cloned_state.place(i, j)
38
39                     if cloned_state.check_is_game_over():
40                         return True, [i, j]
41         return False, None
42
43     def hardcoded_strategy_get_piece(self, state):
44         '''
45         Returns a piece to be placed on the board
46         '''
47         possible_pieces = []
48         for i in range(16):
49             # check if the piece is a winning piece
50             winning_piece, _ = self.check_if_winning_piece(state, i)
51             if (not winning_piece) and (i not in
52                 ↪ list(itertools.chain.from_iterable(state.state_as_array())) and
53                 ↪ (i != state.get_selected_piece())):
54                 possible_pieces.append(i)

```

```

54     # if no pieces can be placed on board anymore (board full/game over),
55     ↪ return -1
56     if len(possible_pieces) == 0:
57         # check if number of non-empty cells is 16
58         if len([i for i in
59             ↪ list(itertools.chain.from_iterable(state.state_as_array())) if i
60             ↪ != -1]) == 16:
61             return -1
62         else:
63             # there are possible pieces to be placed, but they are winning
64             ↪ pieces/already in board
65             on_board = list(itertools.chain.from_iterable(
66                 state.state_as_array()))
67             not_on_board = list(set(range(16)) - set(on_board))
68             return random.choice(not_on_board)
69     else:
70         return random.choice(possible_pieces)
71
72 def choose_piece(self):
73     '''
74     Returns a piece to be placed on the board
75     '''
76     return self.hardcoded_strategy_get_piece()
77
78 def hardcoded_strategy_get_move(self, return_winning_piece_boolean=True):
79     # 1. Play the piece handed over by the opponent:
80     # (a) play a winning position if handed a winning piece;
81     # (b) otherwise, play to build a line of like pieces if possible;
82     # (c) otherwise, play randomly.
83     # 2. Hand a piece to the opponent:
84     # (a) avoid handing over a winning piece for your opponent to play;
85     # (b) otherwise, choose randomly.
86
87     state = self.get_game()
88
89     board = state.state_as_array()
90     selected_piece = state.get_selected_piece()
91     # check if the selected piece is a winning piece
92     winning_piece, position = self.check_if_winning_piece(
93         state, selected_piece)
94     if winning_piece:
95         return selected_piece, position
96
97     # check if the selected piece can be used to build a line of like pieces
98
99     row_1 = [[0, 0], [0, 1], [0, 2], [0, 3]]
100    # pieces in row 2
101    row_2 = [[1, 0], [1, 1], [1, 2], [1, 3]]
102    # pieces in row 3
103    row_3 = [[2, 0], [2, 1], [2, 2], [2, 3]]

```

```

100     # pieces in row 4
101     row_4 = [[3, 0], [3, 1], [3, 2], [3, 3]]
102
103     # pieces in column 1
104     col_1 = [[0, 0], [1, 0], [2, 0], [3, 0]]
105     # pieces in column 2
106     col_2 = [[0, 1], [1, 1], [2, 1], [3, 1]]
107     # pieces in column 3
108     col_3 = [[0, 2], [1, 2], [2, 2], [3, 2]]
109     # pieces in column 4
110     col_4 = [[0, 3], [1, 3], [2, 3], [3, 3]]
111
112     # pieces in diagonal 1
113     diag_1 = [[0, 0], [1, 1], [2, 2], [3, 3]]
114     # pieces in diagonal 2
115     diag_2 = [[0, 3], [1, 2], [2, 1], [3, 0]]
116
117     for line in [row_1, row_2, row_3, row_4, col_1, col_2, col_3, col_4,
118 ↪ diag_1, diag_2]:
119         # check if the selected piece can be used to build a line of like
120         ↪ 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_characteristics(piece)
128                 characteristics.append(
129                     [piece_char.HIGH, piece_char.COLOURED, piece_char.SOLID,
130 ↪ piece_char.SQUARE])
131             else:
132                 empty_rows.append(el)
133                 characteristics.append([-1, -1, -1, -1])
134
135     selected_piece_char = state.get_piece_characteristics(
136         selected_piece)
137     selected_piece_char = [selected_piece_char.HIGH,
138 ↪ selected_piece_char.COLOURED,
139                             selected_piece_char.SOLID,
140 ↪ selected_piece_char.SQUARE]
141
142     # check if characteristics has an empty row
143     if [-1, -1, -1, -1] in characteristics:
144         # insert the selected piece in the empty row
145         empty_piece_index = characteristics.index(
146             [-1, -1, -1, -1])
147         characteristics[empty_piece_index] = selected_piece_char
148
149     # check if any column has the same characteristics

```

```

145         col1 = [characteristics[0][0], characteristics[1][0],
146                 characteristics[2][0], characteristics[3][0]]
147         col2 = [characteristics[0][1], characteristics[1][1],
148                 characteristics[2][1], characteristics[3][1]]
149         col3 = [characteristics[0][2], characteristics[1][2],
150                 characteristics[2][2], characteristics[3][2]]
151         col4 = [characteristics[0][3], characteristics[1][3],
152                 characteristics[2][3], characteristics[3][3]]
153
154         col1 = [int(i) for i in col1]
155         col2 = [int(i) for i in col2]
156         col3 = [int(i) for i in col3]
157         col4 = [int(i) for i in col4]
158
159         if len(set(col1)) == 1 or len(set(col2)) == 1 or len(set(col3))
160         ↪ == 1 or len(set(col4)) == 1:
161             # this piece can be used to build a line of like pieces
162             logging.debug('playing to build a line of like pieces')
163             if return_winning_piece_boolean:
164                 return True, list(reversed(empty_rows[-1]))
165             else:
166                 move = list(reversed(empty_rows[-1]))
167                 return move[0], move[1]
168
169         # play randomly
170         for i in range(self.BOARD_SIDE):
171             for j in range(self.BOARD_SIDE):
172                 for next_piece in range(16):
173                     if state.check_if_move_valid(selected_piece, i, j,
174                     ↪ next_piece):
175                         if return_winning_piece_boolean:
176                             return False, [i, j]
177                         else:
178                             return i, j
179
180         logging.debug(f"Selected piece: {selected_piece}")
181         logging.debug(f"Board: {board}")
182         logging.debug('no move found')
183
184     def place_piece(self):
185         '''
186         Above function sometimes necessary to return additional information
187         In game, first return value is not necessary
188         '''
189
190         return
191         ↪ self.hardcoded_strategy_get_move(return_winning_piece_boolean=False)

```

5.3.3 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]
28                             ↪ for k, v in obj.children.items()},
29
30             # 'children': [NodeEncoder().default(child) for child in
31                             ↪ obj.children],
32             'epsilon': obj.epsilon,
33         }
34         return l
35
36     def encode(self, obj):
37         return super().encode(obj)
38
39     def load_json(self, filename):
40         with open(filename, 'r') as f:
41             return json.load(f, cls=MonteCarloTreeSearchDecoder)
42
43 class MonteCarloTreeSearchDecoder(json.JSONDecoder):
44     """
```

```

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(hashd_state=k)] = [
56             NodeDecoder().object_hook(node) for node in v]
57
58     if 'Q' in obj:
59         return MonteCarloTreeSearch(
60             Q={Node(hashd_state=k): v for k, v in obj['Q'].items()},
61             N={Node(hashd_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):
73     '''
74     Solve using Monte Carlo Tree Search
75     '''
76
77     def __init__(self, board=Quarto(), epsilon=0.1, max_depth=1000, Q=None,
78         ↪ N=None, children=None):
79         self.epsilon = epsilon
80         self.max_depth = max_depth
81         if Q is None:
82             self.Q = defaultdict(int)
83         else:
84             self.Q = defaultdict(int, Q)
85         if N is None:
86             self.N = defaultdict(int)
87         else:
88             self.N = defaultdict(int, N)
89         if children is None:
90             self.children = dict()
91         else:
92             self.children = children
93         self.MAX_PIECES = 16

```



```

93     self.BOARD_SIDE = 4
94     self.board = board
95     self.random_factor = 0
96     self.decisions = 0
97     super().__init__(board)
98
99     def set_board(self, board):
100         self.board = board
101
102     def choose(self, node):
103         '''
104         Choose best successor of node (move)
105         Returns the board itself
106         '''
107         def score(n):
108             logging.debug(f"Before reading in choose {n}")
109             if self.N[n] == 0:
110                 return float('-inf')
111             return self.Q[n] / self.N[n]
112
113         # node is board Quarto
114         node = Node(node)
115         if node.is_terminal():
116             logging.debug(node.board.state_as_array())
117             raise RuntimeError("choose called on terminal node")
118
119         # number of moves made in game
120         self.decisions += 1
121
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(),
130                 ↪ key.board.state_as_array()):
131                     if key in self.children:
132                         print("found in symmetry")
133                         return max(self.children[key], key=score).board
134
135         # number of times have to resort to random
136         rand_child = node.find_random_child()
137         # add to children
138         self.children[node] = [rand_child]
139         return rand_child.board
140
141     print("found in board")
142     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     '''
171     node = Node(board=self.board,
172                 selected_piece_index=self.board.get_selected_piece())
173
174     if node.is_terminal():
175         logging.debug(node.board.state_as_array())
176         raise RuntimeError("choose called on terminal node")
177
178     # if node not in self.children:
179     #     piece, x, y, next_piece = node.find_random_child().move
180     #     # print("Random child")
181     #     # print(piece, x, y, next_piece)
182     #     return x, y, next_piece
183
184     if node not in self.children:
185         for key, value in self.children.items():
186             if BoardTransforms().compare_boards(node.board.state_as_array(),
187                                                  ↪ key.board.state_as_array()):
188                 if key in self.children:
189                     print("found in symmetry")
190                     return max(self.children[key], key=score).board

```

```

191         # number of times have to resort to random
192         rand_child = node.find_random_child()
193         print("Random child")
194         # add to children
195         return rand_child.board.move
196
197     def score(n):
198         logging.debug(f"Before reading in choose {n}")
199         if self.N[n] == 0:
200             return float('-inf')
201         return self.Q[n] / self.N[n]
202
203         # print("In place piece")
204         # print(max(self.children[node], key=score).move)
205         return max(self.children[node], key=score).move[1:]
206
207     def do_rollout(self, board):
208         '''
209         Rollout from the node for one iteration
210         '''
211         logging.debug("Rollout")
212         # if root node, there is no move
213         node = Node(board, move=())
214         path = self.select(node)
215         leaf = path[-1]
216
217         # expand a leaf only when necessary, i.e., only if I arrive at it during
218         # → selection and if it has already been visited (self.N) but not yet
219         # → expanded (self.children)
220         if leaf in self.N and leaf not in self.children:
221             self.expand(leaf)
222
223         reward = self.simulate(leaf)
224         self.backpropagate(path, reward)
225
226     def select(self, node):
227         '''
228         Select path to leaf node
229         '''
230         path = []
231         while True:
232             path.append(node)
233             if node not in self.children or not self.children[node]:
234                 return path
235             unexplored = self.children[node] - self.children.keys()
236             if unexplored:
237                 n = unexplored.pop()
238                 path.append(n)
239                 return path
240             node = self.uct_select(node)

```

```

239
240 def expand(self, node):
241     # logging.debug('Expanding')
242     if node in self.children:
243         return
244     self.children[node] = node.find_children()
245     # logging.debug('Children: ', self.children[node])
246
247 def simulate(self, node):
248     '''
249     Returns reward for random simulation
250     '''
251     invert_reward = False
252     while True:
253         if node.is_terminal():
254             reward = node.reward()
255
256             return 1 - reward if invert_reward else reward
257         node = node.find_random_child()
258         # invert_reward = not invert_reward
259
260 def backpropagate(self, path, reward):
261     '''
262     Backpropagate reward
263     '''
264     logging.debug('Backpropagating')
265     for node in reversed(path):
266         self.N[node] += 1
267         self.Q[node] += reward
268         # TODO: check if this is correct
269         reward = 1 - reward
270
271 def uct_select(self, node):
272     '''
273     Select a child of node, balancing exploration & exploitation
274     '''
275     assert all(n in self.children for n in self.children[node])
276
277     log_N_vertex = math.log(self.N[node])
278
279     def uct(n):
280         return self.Q[n] / self.N[n] + self.epsilon * math.sqrt(log_N_vertex
281             ↪ / self.N[n])
282
283     return max(self.children[node], key=uct)
284
285 def test_win_rate(self, num_trials=10, rollouts=20):
286     print("Testing win rate")
287     agent_wins = 0
288     opponent_wins = 0

```

```

288     draws = 0
289     for i in range(num_trials):
290         board = Quarto()
291         random_player = RandomPlayer(board)
292         self.board = board
293         board.set_selected_piece(random_player.choose_piece(board))
294         while True:
295             # random player moves
296             chosen_location = random_player.place_piece(
297                 board, board.get_selected_piece())
298             chosen_piece = random_player.choose_piece(board)
299             while not board.check_if_move_valid(board.get_selected_piece(),
300                 ↪ chosen_location[0], chosen_location[1], chosen_piece):
301                 chosen_location = random_player.place_piece(
302                     board, board.get_selected_piece())
303                 chosen_piece = random_player.choose_piece(board)
304             board.select(board.get_selected_piece())
305             board.place(chosen_location[0], chosen_location[1])
306             # setting the piece for the next player
307             board.set_selected_piece(chosen_piece)
308             board.switch_player()
309
310             if board.check_is_game_over():
311                 if 1 - board.check_winner() == 0:
312                     opponent_wins += 1
313                 else:
314                     draws += 1
315                 break
316             # monte carlo tree search moves
317
318             # make move with monte carlo tree search
319             for _ in range(rollouts):
320                 self.do_rollout(board)
321                 board = self.choose(board)
322
323             if board.check_is_game_over():
324                 # TODO: check if it's a draw
325                 if 1 - board.check_winner() == 1:
326                     agent_wins += 1
327                 else:
328                     draws += 1
329                 break
330             # don't need to switch player because it's done in choose
331             # random_player needs to do it because it is not done
332             ↪ automatically
333
334     print(f"Agent wins: {agent_wins}/{i+1}")
335     print(f"Random factor ", self.random_factor / self.decisions)
336     self.random_factor = 0
337     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(),
353                 ↪ chosen_location[0], chosen_location[1], chosen_piece):
354                 chosen_location = random_player.place_piece(
355                     board, board.get_selected_piece())
356                 chosen_piece = random_player.choose_piece(board)
357             board.select(board.get_selected_piece())
358             board.place(chosen_location[0], chosen_location[1])
359             # setting the piece for the next player
360             board.set_selected_piece(chosen_piece)
361             board.switch_player()
362
363             if board.check_is_game_over():
364                 if 1 - board.check_winner() == 0:
365                     logging.info("Random player won")
366                 else:
367                     logging.info("Draw")
368                 break
369             # monte carlo tree search moves
370
371             # make move with monte carlo tree search
372             for _ in range(20):
373                 self.do_rollout(board)
374             board = self.choose(board)
375
376             if board.check_is_game_over():
377                 # TODO: check if it's a draw
378                 if 1 - board.check_winner() == 1:
379                     logging.info("Agent won")
380                 else:
381                     logging.info("Draw")
382                 break
383             # don't need to switch player because it's done in choose
384             # random_player needs to do it because it is not done
385             ↪ automatically

```

```

384
385         if i % 2 == 0:
386             # run a test to see if the agent is improving
387             self.test_win_rate()
388
389         # save progress every 10 iterations
390         if i % 100 == 0:
391             logging.debug("Saving progress")
392             if save_format == 'json':
393                 self.save_progress_json('/Volumes/USB/progress3.json')
394             else:
395                 self.save_progress_pickle('progress.pkl')
396
397     def train(self):
398         '''
399         Train without multithreading
400         '''
401         self.train_engine(Quarto(), 100, 'json')
402
403     def threaded_training(self, num_threads=1, save_format='json'):
404         '''
405         Train the model
406         '''
407         thread_pool = []
408
409         for i in range(num_threads):
410             t = Thread(target=self.train_engine, args=(Quarto(), 100, 'json'))
411             t.start()
412             thread_pool.append(t)
413
414         for t in thread_pool:
415             t.join()
416
417         # final save after training
418         if save_format == 'json':
419             self.save_progress_json('progress.json')
420         else:
421             self.save_progress_pickle('progress.pkl')
422
423     def generate_future_probabilities(self, root: Node, node: Node):
424         '''

```

```

425         play an action  $a$  from the root state  $st$  with probability proportional to
↳ the number of times that action was chosen during Phase One. To do this,
↳ AlphaGo Zero creates a probability distribution  $t$  over the actions from the
↳ state  $st$  such that  $t(a) \propto N(st,a)^{-1/}$  for some hyperparameter  $\tau$ ; when  $\tau = 1$  the
↳ distribution exactly matches the ratios of the visit counts, while when  $\tau \rightarrow 0$ 
↳ the probability mass focuses on the action that was chosen most often. Using
↳ this distribution to select actions improves the performance of AlphaGo Zero
↳ because  $t$  is a refinement of the prediction  $pt$  for the start state  $st$ ; as
↳ MCTS is allowed to run, it starts selecting actions with high value estimates
↳ more frequently rather than relying on the prior probability bonus
↳ exploration term.
426         '''
427         # 1 is the default value, but it can be changed to 0.5 or 0.1
428
429         self.tau = 0.5
430         if node not in self.children:
431             self.do_rollout(root.board)
432
433         probs = [self.N[child] / self.N[root]
434                  for child in self.children[node]]
435
436         probs = [p ** (1 / self.tau) for p in probs]
437
438         probs = [p / sum(probs) for p in probs]
439
440         return probs
441
442     def save_progress_pickle(self, filename):
443         with open(filename, 'wb') as f:
444             pickle.dump(self, f)
445
446     def save_progress_json(self, filename):
447         with open(filename, 'w') as f:
448             json.dump(self, f, cls=MonteCarloTreeSearchEncoder)
449
450     def load_progress_json(self, filename):
451         with open(filename, 'r') as f:
452             return json.load(f, cls=MonteCarloTreeSearchDecoder)
453
454     def load_progress(self, filename):
455         with open(filename, 'rb') as f:
456             return pickle.load(f)
457
458
459 if __name__ == "__main__":
460     mcts = MonteCarloTreeSearch()
461     # with open('/Volumes/USB/progress3.json', 'r') as f:
462     #     mcts = decode_tree(json.load(f))
463     #     logging.info("Loaded progress")
464     logging.info("Starting training")

```


5.3.4 Q-Learning + MCTS

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

5.3.5 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
16            ↪ board[i, 3] == -1:
17            piece_1 = state.get_piece_characteristics(board[i, 0])
18            piece_2 = state.get_piece_characteristics(board[i, 1])
19            piece_3 = state.get_piece_characteristics(board[i, 2])
20            if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
21                ↪ piece_1.HIGH == True:
22                positions['triplets'] += 1
23                row_done[i] = True
24            if piece_1.HIGH == piece_2.COLOURED and piece_3.COLOURED ==
25                ↪ piece_2.COLOURED and piece_1.COLOURED == True:
26                positions['triplets'] += 1
27                row_done[i] = True
28            if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
29                ↪ piece_2.SQUARE and piece_1.SQUARE == True:
30                positions['triplets'] += 1
31                row_done[i] = True
32            if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
33                ↪ and piece_1.SOLID == True:
34                positions['triplets'] += 1
35                row_done[i] = True
36    if board[i, 0] == -1 and board[i, 1] != -1 and board[i, 2] != -1 and
37        ↪ board[i, 3] != -1:

```

```

32     piece_1 = state.get_piece_characteristics(board[i, 1])
33     piece_2 = state.get_piece_characteristics(board[i, 2])
34     piece_3 = state.get_piece_characteristics(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_characteristics(board[i, 0])
49         piece_2 = state.get_piece_characteristics(board[i, 1])
50         piece_3 = state.get_piece_characteristics(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_characteristics(board[i, 0])
65         piece_2 = state.get_piece_characteristics(board[i, 2])
66         piece_3 = state.get_piece_characteristics(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_characteristics(board[i, 0])
82         piece_2 = state.get_piece_characteristics(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_characteristics(board[i, 1])
93         piece_2 = state.get_piece_characteristics(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_characteristics(board[i, 2])
104        piece_2 = state.get_piece_characteristics(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
117 # Check all the columns
118
119 for i in range(state.BOARD_SIDE):
120     if board[0, i] != -1 and board[1, i] != -1 and board[2, i] != -1 and
        ↳ board[3, i] == -1:
121         piece_1 = state.get_piece_characteristics(board[0, i])
122         piece_2 = state.get_piece_characteristics(board[1, i])
123         piece_3 = state.get_piece_characteristics(board[2, i])
124         if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↳ piece_1.HIGH == True:
125             positions['triplets'] += 1
126             column_done[i] = True
127         if piece_1.HIGH == piece_2.COLOURED and piece_3.COLOURED ==
            ↳ piece_2.COLOURED and piece_1.COLOURED == True:
128             positions['triplets'] += 1
129             column_done[i] = True
130         if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
            ↳ piece_2.SQUARE and piece_1.SQUARE == True:
131             positions['triplets'] += 1
132             column_done[i] = True
133         if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
            ↳ and piece_1.SOLID == True:
134             positions['triplets'] += 1
135             column_done[i] = True
136     if board[0, i] == -1 and board[1, i] != -1 and board[2, i] != -1 and
        ↳ board[3, i] != -1:
137         piece_1 = state.get_piece_characteristics(board[1, i])
138         piece_2 = state.get_piece_characteristics(board[2, i])
139         piece_3 = state.get_piece_characteristics(board[3, i])
140         if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
            ↳ piece_1.HIGH == True:
141             positions['triplets'] += 1
142             column_done[i] = True
143         if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
            ↳ piece_2.COLOURED and piece_1.COLOURED == True:
144             positions['triplets'] += 1
145             column_done[i] = True
146         if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE ==
            ↳ piece_2.SQUARE and piece_1.SQUARE == True:
147             positions['triplets'] += 1
148             column_done[i] = True
149         if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID
            ↳ and piece_1.SOLID == True:
150             positions['triplets'] += 1
151             column_done[i] = True
152     if board[0, i] != -1 and board[1, i] == -1 and board[2, i] != -1 and
        ↳ board[3, i] != -1:

```

```

153     piece_1 = state.get_piece_characteristics(board[0, i])
154     piece_2 = state.get_piece_characteristics(board[2, i])
155     piece_3 = state.get_piece_characteristics(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_characteristics(board[0, i])
170         piece_2 = state.get_piece_characteristics(board[1, i])
171         piece_3 = state.get_piece_characteristics(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_characteristics(board[0, i])
187         piece_2 = state.get_piece_characteristics(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
196     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:
197         piece_1 = state.get_piece_characteristics(board[1, i])
198         piece_2 = state.get_piece_characteristics(board[2, i])
199         if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
200             positions['couples'] += 1
201         if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
202             positions['couples'] += 1
203         if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
204             positions['couples'] += 1
205         if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
206             positions['couples'] += 1
207     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:
208         piece_1 = state.get_piece_characteristics(board[2, i])
209         piece_2 = state.get_piece_characteristics(board[3, i])
210         if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
211             positions['couples'] += 1
212         if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:
213             positions['couples'] += 1
214         if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
215             positions['couples'] += 1
216         if piece_1.SOLID == piece_2.SOLID and piece_1.SOLID == True:
217             positions['couples'] += 1
218
219     # Array for checking if a certain diagonal already contains a triplet
220     diagonal_done = [False, False]
221
222     # Check first diagonal
223
224     if board[0, 0] != -1 and board[1, 1] != -1 and board[2, 2] != -1 and board[3,
    ↪ 3] == -1:
225         piece_1 = state.get_piece_characteristics(board[0, 0])
226         piece_2 = state.get_piece_characteristics(board[1, 1])
227         piece_3 = state.get_piece_characteristics(board[2, 2])
228         if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
    ↪ piece_1.HIGH == True:
229             positions['triplets'] += 1
230             diagonal_done[0] = True
231         if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
    ↪ piece_2.COLOURED and piece_1.COLOURED == True:
232             positions['triplets'] += 1
233             diagonal_done[0] = True
234         if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
    ↪ and piece_1.SQUARE == True:
235             positions['triplets'] += 1
236             diagonal_done[0] = True

```

```

237     if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
    ↪ piece_1.SOLID == True:
238         positions['triplets'] += 1
239         diagonal_done[0] = True
240 if board[0, 0] == -1 and board[1, 1] != -1 and board[2, 2] != -1 and board[3,
    ↪ 3] != -1:
241     piece_1 = state.get_piece_characteristics(board[1, 1])
242     piece_2 = state.get_piece_characteristics(board[2, 2])
243     piece_3 = state.get_piece_characteristics(board[3, 3])
244     if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
    ↪ piece_1.HIGH == True:
245         positions['triplets'] += 1
246         diagonal_done[0] = True
247     if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
    ↪ piece_2.COLOURED and piece_1.COLOURED == True:
248         positions['triplets'] += 1
249         diagonal_done[0] = True
250     if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
    ↪ and piece_1.SQUARE == True:
251         positions['triplets'] += 1
252         diagonal_done[0] = True
253     if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
    ↪ piece_1.SOLID == True:
254         positions['triplets'] += 1
255         diagonal_done[0] = True
256 if board[0, 0] != -1 and board[1, 1] == -1 and board[2, 2] != -1 and board[3,
    ↪ 3] != -1:
257     piece_1 = state.get_piece_characteristics(board[0, 0])
258     piece_2 = state.get_piece_characteristics(board[2, 2])
259     piece_3 = state.get_piece_characteristics(board[3, 3])
260     if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
    ↪ piece_1.HIGH == True:
261         positions['triplets'] += 1
262         diagonal_done[0] = True
263     if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
    ↪ piece_2.COLOURED and piece_1.COLOURED == True:
264         positions['triplets'] += 1
265         diagonal_done[0] = True
266     if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
    ↪ and piece_1.SQUARE == True:
267         positions['triplets'] += 1
268         diagonal_done[0] = True
269     if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
    ↪ piece_1.SOLID == True:
270         positions['triplets'] += 1
271         diagonal_done[0] = True
272 if board[0, 0] != -1 and board[1, 1] != -1 and board[2, 2] == -1 and board[3,
    ↪ 3] != -1:
273     piece_1 = state.get_piece_characteristics(board[0, 0])
274     piece_2 = state.get_piece_characteristics(board[1, 1])

```



```

275     piece_3 = state.get_piece_characteristics(board[3, 3])
276     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_characteristics(board[0, 0])
291         piece_2 = state.get_piece_characteristics(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_characteristics(board[1, 1])
302         piece_2 = state.get_piece_characteristics(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_characteristics(board[2, 2])
313         piece_2 = state.get_piece_characteristics(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,
326 ↪ 0] == -1:
327         piece_1 = state.get_piece_characteristics(board[0, 3])
328         piece_2 = state.get_piece_characteristics(board[1, 2])
329         piece_3 = state.get_piece_characteristics(board[2, 1])
330         if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
331 ↪ piece_1.HIGH == True:
332             positions['triplets'] += 1
333             diagonal_done[1] = True
334         if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
335 ↪ piece_2.COLOURED and piece_1.COLOURED == True:
336             positions['triplets'] += 1
337             diagonal_done[1] = True
338         if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
339 ↪ and piece_1.SQUARE == True:
340             positions['triplets'] += 1
341             diagonal_done[1] = True
342         if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
343 ↪ piece_1.SOLID == True:
344             positions['triplets'] += 1
345             diagonal_done[1] = True
346     if board[0, 3] == -1 and board[1, 2] != -1 and board[2, 1] != -1 and board[3,
347 ↪ 0] != -1:
348         piece_1 = state.get_piece_characteristics(board[1, 2])
349         piece_2 = state.get_piece_characteristics(board[2, 1])
350         piece_3 = state.get_piece_characteristics(board[3, 0])
351         if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
352 ↪ piece_1.HIGH == True:
353             positions['triplets'] += 1
354             diagonal_done[1] = True
355         if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
356 ↪ piece_2.COLOURED and piece_1.COLOURED == True:
357             positions['triplets'] += 1
358             diagonal_done[1] = True
359         if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
360 ↪ and piece_1.SQUARE == True:
361             positions['triplets'] += 1
362             diagonal_done[1] = True
363         if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
364 ↪ piece_1.SOLID == True:
365             positions['triplets'] += 1
366             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_characteristics(board[0, 3])
359     piece_2 = state.get_piece_characteristics(board[1, 2])
360     piece_3 = state.get_piece_characteristics(board[3, 0])
361     if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
↪ piece_1.HIGH == True:
362         positions['triplets'] += 1
363         diagonal_done[1] = True
364     if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
↪ piece_2.COLOURED and piece_1.COLOURED == True:
365         positions['triplets'] += 1
366         diagonal_done[1] = True
367     if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
↪ and piece_1.SQUARE == True:
368         positions['triplets'] += 1
369         diagonal_done[1] = True
370     if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
↪ piece_1.SOLID == True:
371         positions['triplets'] += 1
372         diagonal_done[1] = True
373 if board[0, 3] != -1 and board[1, 2] == -1 and board[2, 1] != -1 and board[3,
↪ 0] != -1:
374     piece_1 = state.get_piece_characteristics(board[0, 3])
375     piece_2 = state.get_piece_characteristics(board[2, 1])
376     piece_3 = state.get_piece_characteristics(board[3, 0])
377     if piece_1.HIGH == piece_2.HIGH and piece_3.HIGH == piece_2.HIGH and
↪ piece_1.HIGH == True:
378         positions['triplets'] += 1
379         diagonal_done[1] = True
380     if piece_1.COLOURED == piece_2.COLOURED and piece_3.COLOURED ==
↪ piece_2.COLOURED and piece_1.COLOURED == True:
381         positions['triplets'] += 1
382         diagonal_done[1] = True
383     if piece_1.SQUARE == piece_2.SQUARE and piece_3.SQUARE == piece_2.SQUARE
↪ and piece_1.SQUARE == True:
384         positions['triplets'] += 1
385         diagonal_done[1] = True
386     if piece_1.SOLID == piece_2.SOLID and piece_3.SOLID == piece_2.SOLID and
↪ piece_1.SOLID == True:
387         positions['triplets'] += 1
388         diagonal_done[1] = True
389 # Before checking a diagonal for couples, control the boolean flag
390 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:
391     piece_1 = state.get_piece_characteristics(board[0, 3])
392     piece_2 = state.get_piece_characteristics(board[1, 2])
393     if piece_1.HIGH == piece_2.HIGH and piece_1.HIGH == True:
394         positions['couples'] += 1
395     if piece_1.COLOURED == piece_2.COLOURED and piece_1.COLOURED == True:

```

```

396         positions['couples'] += 1
397     if piece_1.SQUARE == piece_2.SQUARE and piece_1.SQUARE == True:
398         positions['couples'] += 1
399     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_characteristics(board[1, 2])
403         piece_2 = state.get_piece_characteristics(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_characteristics(board[2, 1])
414         piece_2 = state.get_piece_characteristics(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']

```

5.4 Code for 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
17 from lib.players import RandomPlayer
18 from tensorflow.keras.models import Sequential, load_model
19 from tensorflow.keras.layers import Dense, Conv2D, MaxPooling2D, Flatten
20 from tensorflow.keras.optimizers import Adam
21 from tensorflow.keras.initializers import HeUniform
22
23 from quarto.objects import Quarto
24
25 env = QuartoScape()
26
27
28 def test(agent):
29     dq_wins = 0
30     for round in range(100):
31         game = Quarto()
32         agent.set_game(game)
33         game.set_players((RandomPlayer(game), agent))
34         winner = game.run()
35         if winner == 1:
36             dq_wins += 1
37         # logging.warning(f"main: Winner: player {winner}")
38     logging.warning(f"main: DQ wins: {dq_wins}")
39
40
41 class DQNAgent:
42     '''Play Quarto using a Deep Q-Network'''
43
44     def __init__(self, env=env, game=None):
45         self.env = env
46         # main model updated every x steps
47         self.model = self._build_model()
48         # target model updated every y steps
49         self.target_model = self._build_model()
50         self.gamma = 0.618
51         self.min_replay_size = 500
52         self.lr = 0.7
53         self.epsilon = 0.8
54         if game is not None:
55             self.env.game = game
56
57         if os.path.exists('model.h5'):
58             # print('Loading model')
59             self.model = tf.keras.models.load_model('model.h5')
60
61     def set_game(self, game):
62         self.env.game = game
63
64     def get_all_actions(self):

```

```

65     '''
66     Return tuples from (0, 0, 0) to (3, 3, 15)
67     Element 1 is position x
68     Element 2 is position y
69     Element 3 is piece chosen for next player
70     '''
71     tuples = []
72     for i in range(0, 4):
73         for j in range(0, 4):
74             for k in range(0, 16):
75                 tuples.append((i, j, k))
76     return tuples
77
78     def _build_model(self):
79         '''
80         Architecture of network:
81         Input nodes are the state of the board
82         Output nodes are the Q-values for each potential action (each output node
↪ is an action)
83         An action is made up of (x, y, piece chosen for next player)
84         There are 16 * 16 * 16 possible actions and the mapping is found in
↪ get_all_actions()
85         '''
86         model = Sequential()
87         initializer = HeUniform()
88         model.add(Dense(
89             12, input_dim=self.env.observation_space.shape[0], activation='relu',
↪ kernel_initializer=initializer))
90         model.add(Dense(24, activation='relu', kernel_initializer=initializer))
91         model.add(Dense(48, activation='relu', kernel_initializer=initializer))
92         model.add(Dense(96, activation='relu', kernel_initializer=initializer))
93         model.add(Dense(192, activation='relu',
94             kernel_initializer=initializer))
95         model.add(Dense(4 * 4 * 16, activation='linear',
96             kernel_initializer=initializer))
97         model.compile(loss=tf.keras.losses.Huber(), metrics=[
98             'mae', 'mse'], optimizer=Adam(learning_rate=0.001))
99         return model
100
101     def build_conv_model(self):
102         model = Sequential()
103         model.add(Conv2D(32, (3, 3), input_shape=(4, 4, 4), activation='relu'))
104         model.add(MaxPooling2D(pool_size=(2, 2)))
105         model.add(Flatten())
106         model.add(Dense(16, activation='relu'))
107         model.add(Dense(4 * 4 * 16, activation='linear'))
108         model.compile(loss='mse', metrics=[
109             'accuracy'], optimizer=Adam(learning_rate=0.001))
110         return model
111

```

```

112 def get_position(self, element, list):
113     if element in list:
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)
153     current_states = np.array([self.abbellire(state, chosen_piece)
154                               for state, chosen_piece, action, reward,
155                               ↪ new_current_state, done in minibatch])
156     current_qs = self.model.predict(current_states.reshape(batch_size, 17))
157     # new current state + chosen_piece for next player -> action (contains
158     ↪ chosen_piece for next player)
159     new_current_states = np.array([self.abbellire(new_current_state,
160     ↪ action[2])

```

```

157         for state, chosen_piece, action, reward,
158             ↪ new_current_state, done in
159             ↪ minibatch])
160
161 future_qs = self.target_model.predict(
162     new_current_states.reshape(batch_size, 17), verbose=0)
163 # exclude invalid moves from calculation
164 X = []
165 Y = []
166 for index, (current_state, chosen_piece, action, reward,
167     ↪ new_current_state, done) in enumerate(minibatch):
168     if not done:
169         # max_future_q = np.max(future_qs[index])
170         # new_q = reward + self.gamma * max_future_q
171         max_future_q = reward + self.gamma * np.max(future_qs[index])
172     else:
173         # max_future_q = reward
174         max_future_q = reward
175
176     # 0 2 5
177     # 0 + 2 * 4 + 5 * 16 = 85
178     current_qs[index][action[0] + action[1] * 4 + action[2] * 16] = (
179         1 - self.lr) * current_qs[index][action[0] + action[1] * 4 +
180         ↪ action[2] * 16] + self.lr * max_future_q
181
182     X.append(self.abellire(current_state, chosen_piece))
183     Y.append(current_qs[index])
184
185 X = np.array(X).reshape(batch_size, 17)
186 Y = np.array(Y).reshape(batch_size, 4 * 4 * 16)
187 logging.debug(X)
188 logging.debug(Y)
189 self.model.fit(X, Y, batch_size=batch_size,
190     verbose=1, shuffle=True, epochs=1)
191
192 def choose_piece(self, state: Any, piece_chosen_for_you: int):
193     '''Choose piece for the next guy to play'''
194     self.env.game.set_board(state)
195     pred = self.make_prediction(state, piece_chosen_for_you)
196     pred = self.nan_out_invalid_actions(-100, pred)
197     best_action = np.nanargmax(pred)
198     best_action = self.get_all_actions()[best_action]
199     return best_action[2]
200
201 def place_piece(self, state: Any, piece_chosen_for_you: int):
202     '''Choose position to move piece to based on the current state'''
203     self.env.game.set_board(state)
204     pred = self.make_prediction(state, piece_chosen_for_you)
205     pred = self.nan_out_invalid_actions(piece_chosen_for_you, pred)
206     best_action = np.nanargmax(pred)
207     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
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],
215             ↪ action[1], action[2]):
216             prediction[i] = np.nan
217
218     return prediction
219
220 def run(self):
221     '''Run training of agent for x episodes'''
222     # ensure both model and target model have same set of weights at the
223     ↪ start
224     self.target_model.set_weights(self.model.get_weights())
225
226     replay_memory = deque(maxlen=5000)
227     state = self.env.reset()
228     # number of episodes to train for
229     num_episodes = 2000
230
231     steps_to_update_target_model = 0
232
233     for episode in range(num_episodes):
234         if episode % 100 == 0:
235             self.model.save(f'/Volumes/USB/qn_weights.h5')
236
237         total_training_reward = 0
238         print(f'Episode: {episode}')
239         state = self.env.reset()
240         done = False
241         # initialise chosen piece with a random piece
242         # in reality, the opponent will choose a piece for you
243         chosen_piece = random.randint(0, 15)
244         while not done:
245             steps_to_update_target_model += 1
246
247             if random.random() < self.epsilon:
248                 action = self.env.action_space.sample()
249                 while not self.env.game.check_if_move_valid(chosen_piece,
250                     ↪ action[0], action[1], action[2]):
251                     action = self.env.action_space.sample()
252             else:

```



```

250         prediction = self.make_prediction(state, chosen_piece)
251         prediction = self.nan_out_invalid_actions(
252             chosen_piece, prediction)
253         if np.all(np.isnan(prediction)):
254             action = self.env.action_space.sample()
255             while not self.env.game.check_if_move_valid(chosen_piece,
256                 ↪ action[0], action[1], action[2]):
257                 action = self.env.action_space.sample()
258         else:
259             action = np.nanargmax(prediction)
260             # get action at index of action
261             action = self.get_all_actions()[action]
262
263         new_state, reward, done, _ = self.env.step(
264             action, chosen_piece)
265
266         replay_memory.append(
267             (state, chosen_piece, action, reward, new_state, done))
268
269         if done:
270             logging.debug('GAME OVER')
271
272         if steps_to_update_target_model % 4 == 0 or done:
273             self.train(replay_memory, 32)
274
275         state = new_state
276         total_training_reward += reward
277
278         if done:
279             total_training_reward += 1
280
281         if steps_to_update_target_model >= 100:
282             self.target_model.set_weights(self.model.get_weights())
283             steps_to_update_target_model = 0
284             break
285
286         chosen_piece = action[2]
287
288         if episode % 10 == 0:
289             logging.info(f'Testing win rate after {episode} episodes')
290             test(self)
291
292         self.lr = self.decay_lr(self.lr, 0.0001, episode)
293
294         self.env.close()
295         self.model.save('/Volumes/USB/qn_weights.h5')
296
297     def main():
298         dq_wins = 0
299         for round in range(100):

```

```

299     game = Quarto()
300     dqn_agent = DQNAgent(game=game)
301     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()

```

The code for this project is available on GitHub at <https://github.com/sidharrth2002/ci-quarto-sidharrth>.

6 Conclusion

Ok bye.