

POLITECNICO DI TORINO

01URRSM

Computational Intelligence Final Report

Sidharrth Nagappan

s307031

Contents

1	Introduction	3
2	Lab 1	4
2.1	Solution	4
2.1.1	Naive Greedy	4
2.1.2	Greedy with basic heuristic approximation	4
2.1.3	A* Search Using a Priority Queue	5
2.1.4	A* Search with Fully Connected Graph (Failed Idea)	8
2.2	Results	13
2.3	Acknowledgements	13
2.4	Received Reviews	15
2.5	Given Reviews	17
2.5.1	Shayan	17
2.5.2	Arman	18
3	Lab 2	21
3.1	Solution	21
3.1.1	Representing the problem	21
3.1.2	Assessing Fitness	21
3.2	Results	22
3.2.1	The Case of Mutations	22
3.3	Mutation Functions	23
3.3.1	Flip Mutation	23
3.3.2	Scramble Mutation	24
3.3.3	Swap Mutation	24
3.3.4	Inversion Mutation	24
3.4	Full Code	25
3.5	Acknowledgements	31
3.6	Received Reviews	31
3.7	Given Reviews	33
3.7.1	Erik	33
3.7.2	Karl	37
3.7.3	Ricardo	42
3.7.4	Francesco	47
4	Lab 3	51
4.1	Solution	51
4.1.1	Fixed Rules	51
4.1.2	Evolved Agent Approach 1	57
4.1.3	Evolved Agent Approach 2 (Probability Thresholds)	66
4.1.4	Minmax	69
4.1.5	Reinforcement Learning	72
4.2	Acknowledgements	80
4.3	Received Reviews	80
4.4	Given Reviews	82
4.4.1	Karl	82
4.4.2	Jaco	95

5	Final Project	98
5.1	Acknowledgements	98
5.2	Strategy For Solving the Problem	98
5.2.1	Step 1: Implement and Tune Multiple Search Algorithms	98
5.2.2	Step 2: Analysing the Algorithms	101
5.2.3	Step 3: Implementing the Hybrid Agent	101
5.3	Results	103
5.4	Code of Hybrid Agent and Sub Players	103
5.4.1	Hybrid Player	103
5.4.2	Faster Version of MCTS	112
5.4.3	Hardcoded Strategy	117
5.4.4	Q-Learning + MCTS	122
5.5	Utility Functions	128
5.5.1	OpenAI Gym Environment for Quarto	128
5.6	Code for Unsuccessful Players	129
5.6.1	Slower MCTS With Different Node Structure	129
5.6.2	Deep Q-Network	139
6	Conclusion and Final Considerations	146

1 Introduction

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

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

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

2 Lab 1

2.1 Solution

Lab 1 concerned the combinatorial optimisation of the set cover problem, which is NP-hard. The problem is to find a minimum set of subsets of a given set of subsets such that all elements of the given set are covered. Since a solution cannot be found in 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 $C = [1, 2]$,

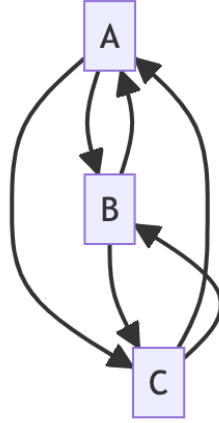


Figure 1: Fully connected graph

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

```

N	w	bloat	visited nodes
5	5	0%	3
10	11	10%	3
50	99	98%	5
100	192	92%	5
500	1313	163%	7
1000	3092	209%	8

Table 1: Smart Greedy (With Heuristic Guessing)

N	w	bloat	visited nodes	visited states
5	5	0%	4	59
10	10	0%	5	191
20	23	15%	934	40216
50	(blow up)	(blow up)	(blow up)	(blow up)

Table 2: A* Traversal

2.2 Results

Results are shown in Tables 1, 2, 3 and 4.

2.3 Acknowledgements

I discussed strategy with Erik Bengtsson (s306792).

N	w	bloat	visited nodes	visited states
5	5	0%	3	34
10	14	40%	4	141
20	35	75%	5	134
50	85	70%	5	134
100	203	103%	6	2127
500	1430	186%	8	12652
1000	3268	227%	9	28941

Table 3: A* Traversal Using Uniform Cost of 1 (Not affected by subset length)

N	w	bloat
5	5	0%
10	10	0%
20	33	65%
50	157	214%
100	297	197%

Table 4: A* Traversal Using a Fully Connected Graph (Possibly Overcomplicating Things)

2.4 Received Reviews

Diego Mangasco

REVIEW BY DIEGO GASCO (DIEGOMANGASCO) SET COVERING (GREEDY): I appreciated a lot the comparison between the professor's Naive greedy approach and your greedy approach! The idea to implement a sort of priority function to choose the best set to add to the solution is nice (a kind of cherry picking). I think you decided to take the set with lowest "f" because you want to keep low the total weight as you can. What if you merge this idea with the number of new elements that the new set can bring to your solution? You can try to find a sort of trade-off between having a new small set and having a new useful one!

SET COVERING (A* TRAVERSAL USING PRIORITY QUEUE): In my implementation I basically used the same approach in developing my A* algorithm! Like you, I decided to implement my heuristics as the number of undiscovered elements, and I took as cost, the length of the new set added in the solution. I also noticed that, with cost sets as unit and not as the length of the new set, the process is much faster, but the solution that we reached is not optimal, so I decided to keep the length as cost.

The only small difference with my implementation is the use of the data structures. To don't have to deal with list manipulation, I preferred to focused my structures in a more set-oriented way. But never mind, these are just personal preferences!

SET COVERING (A* TRAVERSAL USING A FULLY CONNECTED GRAPH) Unfortunately I couldn't try this implementation of A*, because I didn't understand the data structure "adjacency list" and there isn't a block that starts this piece of code like for the previous solutions Reading your explanation about the algorithm idea, I can say that this approach can be useful with a solution space that is not huge, but can become computationally expansive with large N (due to the connections you might have to manage). But anyway with small/medium N it can be helpful in reducing the time of the classical A*.

Ramin

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

Arman

Hi Sid,
here is my review:

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

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

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

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

Bests

2.5 Given Reviews

2.5.1 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.

2.5.2 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 5: 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 results of the algorithm after 1000 generations (only the best results are reported) are shown in Table 5.

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 Acknowledgements

I discussed with Karl Wennerstrom, Diego Gasco and Ricardo Nicida.

3.6 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!

3.7 Given Reviews

3.7.1 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!

3.7.2 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
25 # to initialize the population
26 def create_population(STATE_SPACE, GOAL):
27     """
28     Initialize the population.
29     :param STATE_SPACE: List of lists generated from problem-function
30     :param GOAL: set of integers from 0 to N-1
31     :return: a list of tuples: (genome,fitness), for each individual in the
32     ↪ population.
33     """
34     population = []
35     for _ in range(POPULATION_SIZE):
36         individual = []
37         for _ in range(random.randint(1,len(STATE_SPACE))):
38             l = random.choice(STATE_SPACE)
39             if l not in individual: #check duplicates here
40                 individual.append(l)
41             #individual =
42             ↪ random.choices(STATE_SPACE,k=random.randint(1,len(STATE_SPACE)))
43         fitness = compute_fitness(individual, GOAL)
44         population.append((individual,fitness))
45     return population
46
47 def compute_fitness(genome, GOAL):
48     """
49     fitness is a tuple of (-#of_elems_missing,-#duplicates) which should be
50     ↪ maximized
51     :param genome: the sub-lists with random integers between 0 and N-1
52     :param GOAL: set of integers from 0 to N-1
53     :return: the fitness
54     """
55     # violated constraints, i.e. how many elements are missing
56     vc = GOAL.difference(lists_to_set(genome))
57     duplicates = count_duplicates(genome)
58     # it is worse to lack elements than having duplicates
59     fitness = (-len(vc), -duplicates)
60     return fitness
61
62 def goal_check(genome, GOAL):
63     """
64     Check if all required elements are in the genome
65     :param genome: the sub-lists with random integers between 0 and N-1
66     :param GOAL: set of integers from 0 to N-1
67     :return: boolean value if goal reached or not
68     """

```

```

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

```

```

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(ITERES):
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 ITERES = 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!

3.7.3 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),
48         ↪ k=len(list_of_lists))) for _ in range(POPULATION_SIZE)]
49
50     for genome in genomes:
51         fitness, goal_reached, w = fitness_goal_eval(list_of_lists, genome,
52             ↪ goal)
53         population.append(Individual(genome, fitness, goal_reached, w))
54     return population
55
56 def select_parent(population, tournament_size=2):
57     subset = random.choices(population, k=tournament_size)
58     return max(subset, key=lambda i: i.fitness)
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
64     ↪ to the fitness of the genome
65     ng1 = g1[:cut] + g2[cut:]
66     return ng1
67
68 def mutation(g, genome_size, k=1): # for larger N try to eliminate some of the
69     ↪ 1 in the genome because the bloat was getting to high
70     for _ in range(k):
71         cut = random.randint(1, genome_size)
72         if N < 20:
73             ng = g[:cut-1] + (1-g[cut-1],) + g[cut:]
74         elif N < 500:
75             cut_size = int(genome_size*0.2)
76             new_genome_cut = tuple(random.choices([1, 0], weights=(1, 39),
77                 ↪ k=2*cut_size))
78             ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
79         else:
80             cut_size = int(genome_size*0.2)
81             new_genome_cut = tuple(random.choices([1, 0], weights=(1, 99),
82                 ↪ k=2*cut_size))
83             ng = g[:cut-1-cut_size] + new_genome_cut + g[cut+cut_size:]
84     return ng
85
86 def genetic_algorithm():
87     # create problem
88     list_of_lists = problem(N, seed=42)
89     genome_size = len(list_of_lists)
90     goal = set(range(N))
91
92     # create the population
93     population = generate_population(list_of_lists, goal)

```

```

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!

3.7.4 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)

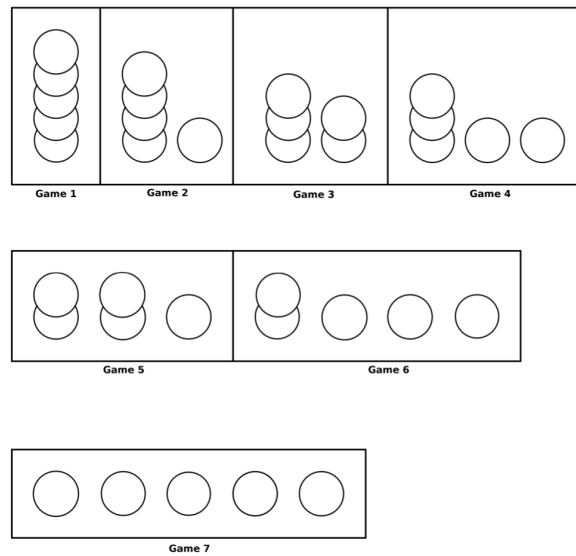
4.1 Solution

4.1.1 Fixed Rules

I came up with multiple rules, through discussion with friends and through this research [paper](#) that define fixed rules for playing Nim. There are currently 4 rules implemented. The rules are as follows:

1. If one pile, take x number of sticks from the pile.
2. If two piles, take x number of sticks from the larger pile.
3. If two piles: a. If 1 pile has 1 stick, take x sticks b. If 2 piles have multiple sticks, take x sticks from the larger pile
4. If three piles and two piles have the same size, remove all sticks from the smallest pile
5. If n piles and n-1 piles have the same size, remove x sticks from the smallest pile until it is the same size as the other piles

Approach 1: A Lot of If-Elses The above rules are applied directly. An if-else sequence decides which strategy to employ based on the current layout and statistics on the nim board.



Player 1 has a winning strategy for all of these games! In game 1, the first player can just take all of the stones immediately. In games 2, 3, 4, and 5, the first player should use his first move to leave his opponent with two piles of the same size, and then mirror the opponents moves for the rest of the game (this will be explained in more detail in exercise 4). In games 6 and 7, the first player should use his first move to leave his opponent with four piles with one stone each; since they each can only take one stone for each of the next four turns, player 1 will win. \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 enumerate(pm)}')
73         else:
74             row = pm[int(index)][0]
75             elems = pm[int(index)][1]
76             break
77     return Nimply(row, elems)
78
79
80 # dumb strategy (to evaluate my agent)
81 def dumb_agent(state: Nim) -> Nimply:
82     """
83     Make stupid move. Always remove 1 from shortest row
84     """
85     data = cook_status(state)
86     row = data['shortest_row']
87     return Nimply(row, 1)
88
89
90 # random strategy (to evaluate my agent)
91 def pure_random(state: Nim) -> Nimply:

```

```

92     """Agent playing completely random"""
93     row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
94     num_objects = random.randint(1, state.rows[row])
95     return Nimply(row, num_objects)
96
97
98 def semi_smart(state: Nim) -> Nimply:
99     """ Make use of rule 1-3, else random"""
100     data = cook_status(state)
101
102     if data['active_rows_number'] == 1:
103         row = data['active_rows_index'][0]
104         elems = state.rows[row]
105         ply = Nimply(row, elems)
106
107     elif data['one_multiple_elem_row']: # all rows but one have a single elem
108         if data['active_rows_number'] % 2 == 0:
109             move = [(r, e) for (r, e) in data["possible_moves"] if state.rows[r]
110                 ↪ - e == 1][0]
111             ply = Nimply(move[0], move[1])
112         else:
113             move = [(r, e) for (r, e) in data["possible_moves"] if
114                 state.rows[r] - e == 0 and r not in
115                 ↪ data['single_elem_rows_index']][0]
116             ply = Nimply(move[0], move[1])
117         else:
118             row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
119             num_objects = random.randint(1, state.rows[row])
120             ply = Nimply(row, num_objects)
121     return ply
122
123 # %% EVOLUTION STRATEGY DESCRIBED
124
125 """
126 (mu, lambda)-strategy
127 1. Create population with the same set of rules but different parameters for
128 ↪ each rule
129 2. k individuals competes in a tournament where the winner becomes a parent
130 3. Perform cross_over between two parents and mutate (aggregate random rule,
131 ↪ e.g. mean(both parents' rule)) with certain prob
132 4. Generate offspring where OFFSPRING_SIZE >> POPULATION_SIZE
133 5. Fitness for offsprings corresponds to how many games are won against their
134 ↪ 'siblings'
135 6. Slice new population from fittest offspring
136 7. Repeat step 2-6 GENERATION times
137 """
138
139 # %% Evolution strategy-functions
140 def init_population():
141     """Initialize population"""
142     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 against
144     ↪ 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, reverse=True)[:POPULATION_SIZE]
174
175
176     # tournament to decide parents
177 def tournament(population: list, k: int) -> dict:
178     """Select best individual out of k competing in a tournament"""
179     contestors = random.sample(population, k=k)
180     best_contestor = sorted(contestors, key=lambda l: l.fitness, reverse=True)[0]
181     return best_contestor
182
183
184 def cross_over(parent1: Evolvable_agent, parent2: Evolvable_agent, mutation_prob:
185     ↪ float) -> Evolvable_agent:

```

```

185     """Generate new individual by cross-over of parents' rules"""
186     rules = [rule for rule in parent1.rules.keys()]
187     new_rules = {}
188     child = Evolvable_agent(NIM_SIZE)
189     for k in rules:
190         which_parent = random.randint(1, 2)
191         new_rules[k] = parent1.rules[k] if which_parent == 1 else
            ↪ parent2.rules[k]
192     if random.random() < mutation_prob:
193         rule = random.choice(rules)
194         if rule == 'rule_1':
195             new_rules[rule] = random.randint(0, (NIM_SIZE - 1) * 2)
196         else:
197             new_rules[rule] = [random.randint(0, 1), random.randint(0, (NIM_SIZE
            ↪ - 1) * 2)]
198     child.rules = new_rules
199     return child
200
201
202 def create_offspring(population: list, k: int, mutation_prob: float) -> list:
203     """Create new offspring"""
204     offspring = []
205     for _ in range(OFFSPRING_SIZE):
206         p1 = tournament(population=population, k=k)
207         p2 = tournament(population=population, k=k)
208         child = cross_over(parent1=p1, parent2=p2, mutation_prob=mutation_prob)
209         offspring.append(child)
210     return offspring
211
212
213 def get_next_generation(offspring: list) -> list:
214     """Find the best individuals in the new generation"""
215     calc_fitness(offspring)
216     return fittest_individuals(offspring)
217
218
219 # %% PLAYING FUNCTIONS
220 def evaluate(strategy1: Callable, strategy2: Callable) -> float:
221     """Play two strategies against each other and evaluate their performance """
222     players = (strategy1, strategy2)
223     won = 0
224
225     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
325         *  $N(v)$  - number of visits for node  $v$ 
326         *  $Q(v)$  - value/reward playing from that node
327
328 UCT:
329     ↪  $uct(v_i, v) = Q(v_i)/N(v_i) + c\sqrt{\log(N(v))/N(v_i)}$ , which prefers child
330     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     def __init__(self, nim_size: int, random_factor=0.2,
342                 exploration_factor=np.sqrt(2)): # explore with 20%, exploit
    ↪ with 80%
343         self.nim_size = nim_size
344         self.current_state = None
345         self.previous_state = None
346         self.__init_states(nim_size)
347         self.random_factor = random_factor
348         self.c = exploration_factor
349
350     def __init_states(self, nim_size: int):
351         """find all possible board positions"""
352         states = {}
353         rows = [i * 2 + 1 for i in range(nim_size)]
354         elem_ranges = list(itertools.combinations([range(n + 1) for n in rows],
    ↪ r=nim_size))
355         all_states = list(itertools.product(*elem_ranges[0]))
356
357         for state in all_states:
358             states[state] = {}
359             states[state]['visits'] = 0
360             states[state]['value'] = 0
361             states[state]['child_states'] = self.__init_child_states(state)
362         self.states = states
363         # last state is the initial board
364         self.current_state = all_states[-1]
365         self.states[self.current_state]['visits'] = 1
366
367     def __init_child_states(self, state):
368         """Find all states accessible from state"""
369         nim = Nim(self.nim_size)
370         nim._rows = list(state)
371         if nim:
372             data = cook_status(nim)
373             children = []
374             for ply in data['possible_moves']:

```

```

375         tmp_nim = deepcopy(nim)
376         tmp_nim.nimming(ply)
377         children.append(tmp_nim.rows)
378     return children
379
380 # MCTS -----
381 def selection(self):
382     """Select next move according to highest uct score"""
383     next_state = self.__state_with_highest_uct()
384     return next_state
385
386 def __state_with_highest_uct(self):
387     """Move to child node with highest UCT score (depending on parent and
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 state"""
418     # manipulate nim
419     nim = Nim(self.nim_size)
420     nim._rows = list(self.previous_state)
421     data = cook_status(nim)
422     ply = [ply for ply in data['possible_moves'] if data['rows'][ply[0]] -
423     ↪ ply[1] == self.current_state[ply[0]]][0]
424     return ply

```

```

420
421 def simulate(self, opponent: Callable, n_matches: int):
422     """Simulate game of nim vs opponent by letting RL agent play randomly
423     ↪ from current state"""
424     players = (opponent, pure_random) # rl agent is second since played move
425     ↪ to get here
426     nim = Nim(self.nim_size)
427     won = 0
428     for match in range(n_matches):
429         # forbidden stuff
430         nim._rows = list(self.current_state) # play from current state
431
432         player = 0
433         while nim:
434             ply = players[player](nim)
435             nim.nimming(ply)
436             player = 1 - player
437         if player == 0:
438             won += 1
439
440     # update results
441     self.backpropagate(n_matches, won)
442
443 def backpropagate(self, visits: int, reward: int):
444     """Update results after simulating `visits` times game from current
445     ↪ state"""
446     self.states[self.current_state]['visits'] += visits
447     self.states[self.current_state]['value'] += reward
448
449 # TRAINING -----
450 def learn_to_play(self, opponents: list, n_sims: int, n_matches: int):
451     """Simulate the game from original state. For each move, simulate the
452     ↪ outcome n_matches times.
453     Keep moving until board is empty, then repeat n_sims times."""
454     for opponent in opponents:
455         for n in tqdm(range(n_sims), desc="Iterations, %s"
456             ↪ %opponent.__name__):
457             # always start from initial state in a new simulation
458             nim = Nim(self.nim_size)
459             self.current_state = nim.rows
460
461             while nim:
462                 if random.random() < self.random_factor:
463                     # choose random state
464                     ns = self.random_selection()
465                 else:
466                     ns = self.selection()
467                 ply = self.expand(next_state=ns)
468                 nim.nimming(ply)
469
470

```

```

465         self.simulate(opponent, n_matches)
466
467     def get_statistics(self):
468         """Print overview of number of visits and wins for a visited state"""
469         info = [(k, v['value'], v['visits']) for k, v in self.states.items()]
470         for state in info:
471             if state[2] > 0: # at least 1 visit
472                 print(f'State {state[0]}: \tvisits {state[2]} \twins {state[1]}')
473
474     def policy(self, state: Nim) -> Nimply:
475         """The policy, i.e. the next move for the current state"""
476         self.current_state = state.rows
477         ns = self.selection()
478         ply = self.expand(next_state=ns)
479         return ply
480
481 # %% MAIN
482 import argparse
483
484 if __name__ == '__main__':
485
486     # VARIABLES
487     NIM_SIZE = 3
488     NUM_MATCHES = 100
489     EVAL_MATCHES = 100
490
491     # INPUT
492     parser = argparse.ArgumentParser()
493     parser.add_argument("-t", "--task", dest="task", default=1,
494                         help="Which task should run? Choose from 1, 2, 3 or 4.",
495                         ↪ type=int)
496
497     args = parser.parse_args()
498     print(f"Task: {args.task}")
499
500     # -----TASK 1 - PLAYING THE OPTIMAL STRATEGY
501     ↪ -----
502     if args.task == 1:
503         play_nim(optimal_strategy, optimal_strategy)
504         # play the nim-sum strategy
505         starting_wins = evaluate(optimal_strategy, optimal_strategy)
506         print(f'Optimal strategy wins {starting_wins * 100: .0f}% when starting
507               ↪ and {(1 - starting_wins) * 100: .0f}% when not starting.')
508
509     # -----TASK 2 - EVOLVE AN AGENT
510     ↪ -----
511     elif args.task == 2:
512         # set params
513         POPULATION_SIZE = 50
514         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     ↪ -----
526 elif args.task == 3:
527     import time
528     start = time.time()
529     play_nim(best_move, optimal_strategy)
530     elapsed = time.time() - start
531     print(f'It take {elapsed :.2f} seconds to play a game of Nim with size
532           ↪ {NIM_SIZE}')
533
534     # ----- TASK 4 - REINFORCEMENT LEARNING
535     ↪ -----
536 elif args.task == 4:
537     ITERS = 1000
538
539     # must have run with -t 2 to have a pop
540     if 'pop' in locals():
541         opponents = [pure_random, semi_smart, make_strategy(pop[0]),
542                     ↪ optimal_strategy]
543     else:
544         opponents = [pure_random, semi_smart, optimal_strategy]
545
546     for opponent in opponents:
547         rl_agent = RLAgent(NIM_SIZE)
548         rl_agent.learn_to_play([opponent], n_sims=ITERS,
549                               ↪ n_matches=NUM_MATCHES)
550         evaluate(rl_agent.policy, opponent)
551
552 else:
553     print(f'Have not finished task {args.task}')

```

Hi Karl,

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

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

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

2. I like that you have the option to play your agents against a human. I wish I also did this, as it's interesting to run.
3. The README is very well written and the code is well documented with comments in the right places. I had no issues understand your rules for the evolvable agent, especially since the rules were both explained and linked to individual lines of code.

Evolutionary Algorithm

1. The rules are neat in the sense that rules 4, 5 and 6 are very generalised and will apply to any setup on the board that does not match rules 1, 2 and 3. Hence, the agent always has something to fall back on, without resorting to a completely random move. However, the rules you implemented are a small subset of a much larger collection in the literature. A few extra rules can be added to cater to very specific scenarios like "one row left with 2 elements", or a compound rule like: "if one row has x elements" and "another row has 1 elements", then "remove 1 element from the last row". I understand that there are an infinite number of possibilities, but hardcoding a few more for a small nim size is harmless.
2. I like that you modularised your agent with different methods for each rule. It really cleans up the 'if-else' series code block. This is something I didn't do and I will take inspiration from keeping the agent as a separate class.
3. Your mutation strategy to average two genome dictionary values instead of simply swapping them is interesting and may result in fewer cases where the mutated value is unusually small/large for a particular rule. I'll definitely take inspiration from this.

Minimax

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

Reinforcement Learning

1. I just learnt about Upper Confidence Trees after reading your code, where it seems to resemble some form of tree search. The best children are identified with RL by running the game from that particular state during learning. All in all, this is very well implemented.
2. My only suggestion is to decay/adjust the `random_factor` during each match. I found that adjusting the exploration epsilon rendered better performance when decayed, favouring exploration at the start and exploitation towards the end. This is just an idea, am not sure how it will work for UCTs.

Overall, good job!

Best, Sidharrth

4.4.2 Jaco

Jaco's code

```
1 tree=None
2
3 def minmax_agent(state: Nim) -> Nimply:
4
5     global tree
6     nodes=[[node for node in children] for children in
7             ↪ LevelOrderGroupIter(tree,maxlevel=2)]
8
9     #CHECK IF TREE IS UP TO DATE
10
11     root=nodes[0][0]
12     root_name=root.name[0]
13     nim_root=Nim(0)
14     nim_root.fromString(root_name)
15     if(state.__eq__(nim_root)):
16         pass
17     else:
18         for i in nodes[1]:
19             F=Nim(0)
20             F.fromString(i.name[0])
21             if(state.__eq__(F)):
22                 i.parent=None
23                 tree=i
24                 break
25
26     #CHECK BEST MOVE
27     nodes=[[node for node in children] for children in
28             ↪ LevelOrderGroupIter(tree,maxlevel=2)]
```



```

29     #Final-move check
30
31     root=nodes[0][0]
32     root_name=root.name[0]
33     nim_root=Nim(0)
34     nim_root.fromString(root_name)
35     if(nim_root.last_move()):
36         for i,j in enumerate(nim_root.rows):
37             if j>0:
38                 return Nimply(i,j)
39
40
41     lower=np.inf
42     lowerNode=None
43
44     for i in nodes[1]:
45         if(i.name[1]<lower):
46             lowerNode=i
47             lower=i.name[1]
48
49     nim_temp=Nim(0)
50
51     nim_temp.fromString(lowerNode.name[0])
52
53     move=state.moveFromOtherNim(nim_temp)
54
55     #update tree
56
57     tree=make_tree(nim_temp)
58
59     '''
60     print("tree2=")
61     print(RenderTree(tree, style=DoubleStyle))
62     print("\n\n")
63     print("-----")
64     '''
65     return move

```

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

1. The README is well-explained, I didn't have much of a problem understanding which strategies were better than others.
2. I also used temporal difference learning as my RLAgent agent for the last task, and I think it is a suitable implementation in this case, as there are not many possible Nim states to consider.

Things to look at:

1. For the GA, I notice that you use a mutation rate of '0.5' that stays constant throughout training. You could consider decaying the value, as I, along with others in the Telegram group, found that high mutation rates at the start were detrimental to training.
2. The computational cost of min-max pruning is vast, so maybe you could consider implementing alpha beta pruning to speed up the process.
3. While your RL agent's implementation is sound, I wonder why your win rate against random is only 48%. You could run for more iterations to see if the win rate improves.

5 Final Project

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

5.1 Acknowledgements

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

- After I tried to get a working Deep Q-Network and realised that it wasn't converging in reasonable time, we discussed the possibility of using some tree search algorithm. It was Diego who suggested MCTS.
- When realising that MCTS can be quite slow towards the end of the game, I suggested building a hybrid QL-MCTS player that would use a base Q-table to remember the best moves so the quadratically complex tree wouldn't need to store so many nodes.
- We later built a hardcoded agent using different rules and realised that it performed very well, and was quick to make a move.
- We then decided to combine everything we did into a hybrid agent that would switch between strategies depending on the board score. He also suggested that a genetic algorithm could be used for this. Diego suggested a good scoring function that could switch between the boards.
- 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 such as board scoring or isomorphic board comparisons.

The code for this project is available on [Github](#).

5.2 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 effi-**

ciency:

- **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. The risky strategy is as follows (from Peter Rowlett's [paper](#)).
 1. Play the piece handed over by the opponent: (a) play a winning position if handed a winning piece; (b) otherwise, play to build a line of like pieces if possible; (c) otherwise, play randomly.
 2. Hand a piece to the opponent: (a) avoid handing over a winning piece for your opponent to play; (b) otherwise, choose randomly.
- **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. I build two variations of Deep Q-networks linear DQN (made up of Dense layers) and Convolutional DQNs (made up of Conv2D layers).
 - The input to the linear network is a flattened list of 1x17 pieces based on the current board composition, and the selected piece.
 - The input to the convolutional neural network is a 5x5x4 board composition, made up of the 3D characteristics of each piece and the selected piece for the player to play. The 5th row and column are appended and replicated with the selected piece for the player to play.

The output of both models is a softmax vector of possible actions (x, y, next_piece). A custom OpenAI Gym environment is created to make training, game steps and rewards easier to manage.

- **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.
- **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. The algorithm from [Geeks for Geeks](#) is shown in Figure 3.

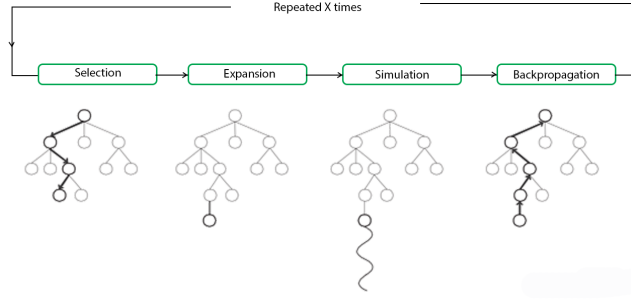


Figure 3: MCTS Algorithm

- **QL-MCTS:** This algorithm uses a Q-table as its base and uses a rolled out Monte Carlo Search Tree for a more efficient search during the training phase. In testing, 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 its 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:

- 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, we 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

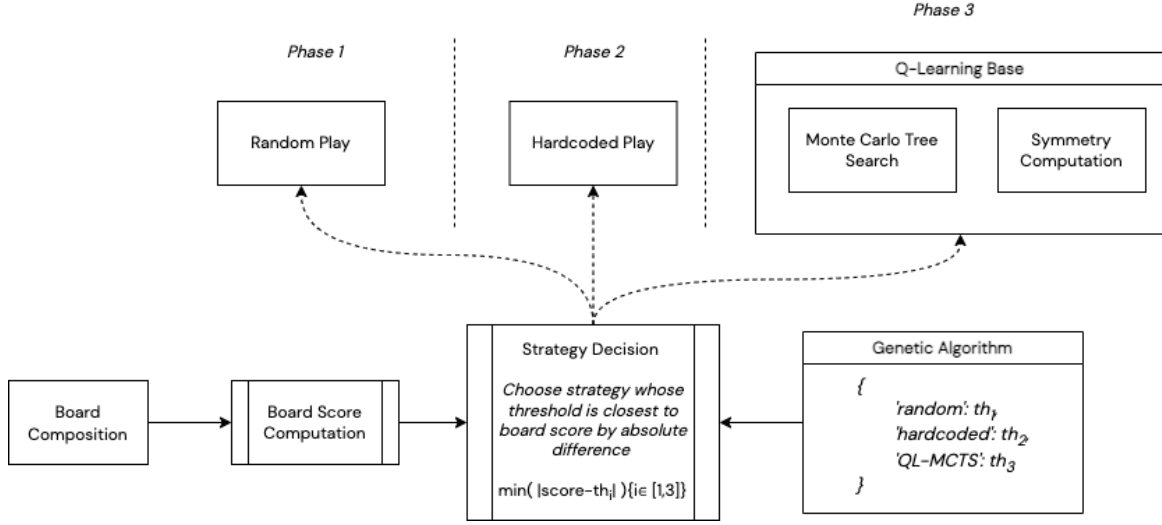


Figure 4: Hybrid Agent

Listing 1 Genome Example

```

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

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 4, and is a balance between speed and efficiency.

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

$$couples + 2 * triplets$$

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

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

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

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

5.3 Results

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

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

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

Tournament results are shown in Table 6, where each player is played against a random player for 10 tournaments of 10 games each (10 x 10 = 100 games).

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

Table 6: Results computed based on 10 tournaments of 10 games each (10 x 10 = 100 games) against a random player

5.4 Code of Hybrid Agent and Sub Players

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

5.4.1 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 os
5  import sys
6  sys.path.insert(0, '..')
7
8  import tqdm
9  import random
10 import logging
11 import json
12 import itertools
13 from copy import deepcopy
14 from lib.players import Player, RandomPlayer
15 from quarto.objects import Quarto
16 from lib.scoring import score_board
17 from QLMCTS import QLearningPlayer
18 from Hardcoded.hardcoded import HardcodedPlayer
19
20 logging.basicConfig(level=logging.DEBUG)
21
22 class Genome:
23     def __init__(self, thresholds, fitness):
24         self.thresholds = thresholds
25         self.fitness = fitness
26
27     def set_fitness(self, fitness):
28         self.fitness = fitness
29
30     def set_thresholds(self, thresholds):
31         self.thresholds = thresholds
32
33     def toJSON(self):
34         return {
35             'thresholds': self.thresholds,
36             'fitness': self.fitness
37         }
38
39
40 class FinalPlayer(Player):
41     '''
42     Final player uses genetic algorithm to decide between:
43     1. Hardcoded Strategy
44     2. Random Strategy
45     3. QL-MCTS
```

```

46     '''
47
48     def __init__(self, quarto: Quarto = None):
49         if quarto is None:
50             quarto = Quarto()
51         super().__init__(quarto)
52         self.ql_mcts = QLearningPlayer(quarto)
53         self.hardcoded = HardcodedPlayer(quarto)
54         self.random_player = RandomPlayer(quarto)
55         self.BOARD_SIDE = 4
56         self.GENOME_VAL_UPPER_BOUND = 16
57         self.GENOME_VAL_LOWER_BOUND = 0
58         self.thresholds = {
59             'random': 1.090773081612301,
60             'hardcoded': 2.790328881747581,
61             'ql-mcts': 6.251997327518943
62         }
63         self.ql_mcts_next_piece = -1
64
65     def generate_population(self, population_size):
66         population = []
67         for i in range(population_size):
68             threshold = {}
69
70             # make sure that value for random < hardcoded < ql-mcts
71             threshold['random'] = random.random() * self.GENOME_VAL_UPPER_BOUND
72             # find random number between random and 15
73             threshold['hardcoded'] = threshold['random'] + \
74                 random.random() * (self.GENOME_VAL_UPPER_BOUND -
75                                     threshold['random'])
76
77             # find random number between hardcoded and 15
78             threshold['ql-mcts'] = threshold['hardcoded'] + \
79                 random.random() * (self.GENOME_VAL_UPPER_BOUND -
80                                     threshold['hardcoded'])
81
82             assert threshold['random'] < threshold['hardcoded'] <
83                 ↪ threshold['ql-mcts']
84
85             population.append(Genome(threshold, 0))
86         return population
87
88     def ensure_correct_ordering(self, new_thresholds):
89         if new_thresholds['random'] > new_thresholds['hardcoded']:
90             new_thresholds['random'], new_thresholds['hardcoded'] =
91                 ↪ new_thresholds['hardcoded'], new_thresholds['random']
92         if new_thresholds['hardcoded'] > new_thresholds['ql-mcts']:
93             new_thresholds['hardcoded'], new_thresholds['ql-mcts'] =
94                 ↪ new_thresholds['ql-mcts'], new_thresholds['hardcoded']
95         if new_thresholds['random'] > new_thresholds['hardcoded']:

```

```

93         new_thresholds['random'], new_thresholds['hardcoded'] =
94             ↪ new_thresholds['hardcoded'], new_thresholds['random']
95     return new_thresholds
96
97 def crossover(self, genome1, genome2):
98     new_thresholds = {}
99     for key in genome1.thresholds:
100         new_thresholds[key] = random.choice(
101             [genome1.thresholds[key], genome2.thresholds[key]])
102
103     # make sure that value for random < hardcoded < ql-mcts
104     new_thresholds = self.ensure_correct_ordering(new_thresholds)
105     return Genome(new_thresholds, 0)
106
107 def mutate(self, genome):
108     new_thresholds = {}
109     genome_thresholds = genome.thresholds
110     if random.random() < 0.4:
111         new_thresholds['random'] = random.random() * \
112             self.GENOME_VAL_UPPER_BOUND
113         new_thresholds['hardcoded'] = random.choice(
114             [genome_thresholds['random'], genome_thresholds['random'] +
115              random.random() * (self.GENOME_VAL_UPPER_BOUND -
116              ↪ genome_thresholds['random'])])
117         new_thresholds['ql-mcts'] = random.choice(
118             [genome_thresholds['hardcoded'], genome_thresholds['hardcoded'] +
119              random.random() * (self.GENOME_VAL_UPPER_BOUND -
120              ↪ genome_thresholds['hardcoded'])])
121
122     new_thresholds = self.ensure_correct_ordering(new_thresholds)
123
124     assert new_thresholds['random'] < new_thresholds['hardcoded'] <
125         ↪ new_thresholds['ql-mcts']
126
127     return Genome(new_thresholds, 0)
128     return genome
129
130 def evolve(self, num_generations=50):
131     self.population_size = 50
132     self.offspring_size = 10
133     population = self.generate_population(self.population_size)
134
135     pbar = tqdm.tqdm(total=num_generations)
136     for gen in range(num_generations):
137         pbar.update(1)
138         logging.debug('Generation: {}'.format(gen))
139         offspring = []
140         for i in range(self.offspring_size):
141             parent1 = random.choice(population)
142             parent2 = random.choice(population)

```

```

139         child = self.crossover(parent1, parent2)
140         child = self.mutate(child)
141         child.fitness = self.play_game(child.thresholds, num_games=5)
142         offpsring.append(child)
143     population += offpsring
144     population = sorted(
145         population, key=lambda x: x.fitness,
146         ↪ reverse=True)[:self.population_size]
147
148     if gen % 5 == 0:
149         logging.info('Saving population')
150         with open('/Volumes/USB/population3.json', 'w') as f:
151             json.dump([genome.toJSON() for genome in population], f)
152
153     # return the best genome's thresholds
154     return population[0].thresholds
155
156 def play_game(self, thresholds, num_games=10):
157     wins = 0
158     for game in range(num_games):
159         logging.debug('Game: {}'.format(game))
160         state = Quarto()
161         player = 0
162
163         # initialise with some random piece just to kickstart game
164         state.set_selected_piece(self.random_player.choose_piece(state, 0))
165         self.current_state = state
166
167         # python passes by reference
168         # agent will use the state, etc. to update the Q-table
169         # this function also wipes the MCTS tree
170         self.ql_mcts.clear_and_set_current_state(state)
171         self.hardcoded = HardcodedPlayer(state)
172
173     while True:
174         # board score is the number of couples and triplets on the board
175         # it is indicative of the change of the board state
176         board_score = score_board(self.current_state)
177
178         differences = [abs(board_score - thresholds[key])
179                        for key in thresholds]
180         min_diff = min(differences)
181         index = differences.index(min_diff)
182         key = list(thresholds.keys())[index]
183
184         if player == 0:
185             if key == 'random':
186                 logging.debug('random')
187                 # play randomly
188                 action = self.random_player.place_piece()

```

```

188         next_piece = self.random_player.choose_piece()
189         while
190             ↪ self.current_state.check_if_move_valid(self.current_state.get_s
191             ↪ action[0], action[1], next_piece) is False:
192                 action = self.random_player.place_piece()
193                 next_piece = self.random_player.choose_piece()
194         self.current_state.select(
195             self.current_state.get_selected_piece())
196         self.current_state.place(action[0], action[1])
197         self.current_state.set_selected_piece(next_piece)
198         self.current_state.switch_player()
199         player = 1 - player
200
201     elif key == 'hardcoded':
202         # play using hardcoded strategy
203         self.previous_state = deepcopy(self.current_state)
204         winning_piece, position =
205             ↪ self.hardcoded.hardcoded_strategy_get_move()
206         next_piece =
207             ↪ self.hardcoded.hardcoded_strategy_get_piece()
208         while
209             ↪ self.current_state.check_if_move_valid(self.current_state.get_s
210             ↪ position[0], position[1], next_piece) is False:
211                 winning_piece, position =
212                     ↪ self.hardcoded.hardcoded_strategy_get_move()
213                 next_piece =
214                     ↪ self.hardcoded.hardcoded_strategy_get_piece()
215         self.current_state.select(state.get_selected_piece())
216         self.current_state.place(position[0], position[1])
217         self.current_state.set_selected_piece(next_piece)
218         self.current_state.switch_player()
219         player = 1 - player
220
221     else:
222         # play using QL-MCTS
223         print('ql-mcts')
224         self.ql_mcts.previous_state = deepcopy(
225             self.current_state)
226         action = self.ql_mcts.get_action(self.current_state)
227         self.ql_mcts.previous_action = action
228         # store the next piece for when choose is called
229         # self.ql_mcts_next_piece =
230             ↪ self.ql_mcts.tree.choose_piece()
231         self.ql_mcts_next_piece =
232             ↪ self.ql_mcts.tree.choose_piece()
233         self.ql_mcts.current_state.select(
234             self.current_state.get_selected_piece())
235         self.ql_mcts.current_state.place(action[0], action[1])
236         self.ql_mcts.current_state.set_selected_piece(
237             self.ql_mcts_next_piece)

```

```

228         self.ql_mcts.current_state.switch_player()
229         player = 1 - player
230
231     else:
232         # opponent is random
233         action = self.random_player.place_piece()
234         next_piece = self.random_player.choose_piece()
235         while
236             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_piece(),
237             ↪ action[0], action[1], next_piece) is False:
238                 action = self.random_player.place_piece()
239                 next_piece = self.random_player.choose_piece()
240                 # WARNING: very often stuck in this loop
241         self.current_state.select(
242             self.current_state.get_selected_piece())
243         self.current_state.place(action[0], action[1])
244         self.current_state.set_selected_piece(next_piece)
245         self.current_state.switch_player()
246         player = 1 - player
247
248     if self.current_state.check_is_game_over():
249         if 1 - self.current_state.check_winner() == 0:
250             print("Agent wins")
251             wins += 1
252             # TODO: QL reward update
253         else:
254             print("Player 2 wins")
255             break
256
257     # fitness is the percentage of games won
258     logging.debug(f"Win rate: {wins/num_games}")
259     return wins/num_games
260
261 def choose_piece(self):
262     """
263     Choose piece for next player to place
264     """
265     thresholds = self.thresholds
266
267     # game is stored in parent
268     self.current_state = self.get_game()
269
270     board_score = score_board(self.current_state)
271
272     differences = [abs(board_score - thresholds[key])
273                   for key in thresholds]
274     min_diff = min(differences)
275     index = differences.index(min_diff)
276     key = list(thresholds.keys())[index]

```

```

276     # python passes by reference
277     # agent will use the state, etc. to update the Q-table
278     # this function also wipes the MCTS tree
279     # self.ql_mcts.clear_and_set_current_state(self.current_state)
280     self.hardcoded = HardcodedPlayer(self.current_state)
281
282     if self.ql_mcts_next_piece != -1:
283         if self.ql_mcts_next_piece not in
284             ↪ list(itertools.chain(*self.current_state.state_as_array())):
285                 print('ql-mcts choose')
286                 return self.ql_mcts_next_piece
287
288     if key == 'random':
289         # play randomly
290         next_piece = self.random_player.choose_piece()
291         while next_piece in
292             ↪ list(itertools.chain(*self.current_state.state_as_array())):
293                 next_piece = self.random_player.choose_piece()
294         self.ql_mcts_next_piece = -1
295         return next_piece
296
297     # elif key == 'hardcoded':
298     else:
299         # play using hardcoded strategy
300         print('hardcoded')
301         self.previous_state = deepcopy(self.current_state)
302         next_piece = self.hardcoded.hardcoded_strategy_get_piece()
303         self.ql_mcts_next_piece = -1
304         return next_piece
305
306     def place_piece(self):
307         # python passes by reference
308         # agent will use the state, etc. to update the Q-table
309         # this function also wipes the MCTS tree
310         self.current_state = self.get_game()
311         thresholds = self.thresholds
312
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.ql_mcts.clear_and_set_current_state(self.current_state)
317
318         self.hardcoded = HardcodedPlayer(self.current_state)
319
320         while True:
321             # board score is the number of couples and triplets on the board
322             # it is indicative of the change of the board state
323             board_score = score_board(self.current_state)
324
325             differences = [abs(board_score - thresholds[key])

```

```

324         for key in thresholds]
325     min_diff = min(differences)
326     index = differences.index(min_diff)
327     key = list(thresholds.keys())[index]
328
329     if key == 'random':
330         logging.debug('random')
331         # play randomly
332         action = self.random_player.place_piece()
333         next_piece = self.random_player.choose_piece()
334         while
335             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_
336             ↪ action[0], action[1], next_piece) is False:
337                 action = self.random_player.place_piece()
338                 next_piece = self.random_player.choose_piece()
339         return action[0], action[1]
340
341     elif key == 'hardcoded':
342         # play using hardcoded strategy
343         logging.debug('hardcoded')
344         self.previous_state = deepcopy(self.current_state)
345         winning_piece, position =
346             ↪ self.hardcoded.hardcoded_strategy_get_move()
347         # next_piece = self.hardcoded_strategy_get_piece()
348         # while
349             ↪ self.current_state.check_if_move_valid(self.current_state.get_selected_
350             ↪ position[0], position[1], next_piece) is False:
351                 # winning_piece, position =
352                 ↪ self.hardcoded_strategy_get_move()
353         # next_piece = self.hardcoded_strategy_get_piece()
354         return position[0], position[1]
355
356     else:
357         # play using QL-MCTS
358         logging.debug('ql-mcts')
359         print(f"Selected piece:
360             ↪ {self.current_state.get_selected_piece()}")
361         self.ql_mcts.previous_state = deepcopy(
362             self.current_state)
363         action = self.ql_mcts.get_action(self.current_state)
364         self.ql_mcts.previous_action = action
365         # store the next piece for when choose is called
366         # self.ql_mcts_next_piece = self.ql_mcts.tree.choose_piece()
367         self.ql_mcts_next_piece = self.ql_mcts.tree.choose_piece()
368         return action[0], action[1]
369
370 def test_thresholds(self):
371     win_rate = self.play_game(self.thresholds, num_games=5)
372     print(f"Win rate: {win_rate}")
373     return win_rate

```



```

367
368 if __name__ == "__main__":
369     final_player = FinalPlayer()
370     average_win_rate = 0
371     for i in range(10):
372         win_rate = final_player.test_thresholds()
373         average_win_rate += win_rate
374     print(f"Average win rate: {average_win_rate}")

```

5.4.2 Faster Version of MCTS

The implementation of MCTS and the rollout strategy is based on the minimal implementation [here](#).

```

1  from copy import deepcopy
2  import hashlib
3  import itertools
4  import os
5  import random
6  import numpy as np
7  from lib.isomorphic import BoardTransforms
8  from quarto.objects import Quarto
9
10 class Node:
11     def __init__(self, state: Quarto = Quarto(), place_current_move=None,
12         ↪ final_point=False):
13         self._state = state
14         self.place_current_move = place_current_move
15         self.final_point = final_point
16         self.wins = 0
17         self.visits = 0
18
19     def __hash__(self):
20         string = str(self._state.get_selected_piece()) +
21         ↪ np.array2string(self._state.state_as_array())
22         return int(hashlib.sha1(string.encode('utf-8')).hexdigest(), 32)
23
24     def __eq__(self, other):
25         if not isinstance(other, Node):
26             return False
27         return np.array_equal(self._state.state_as_array(),
28         ↪ other._state.state_as_array()) and self._state.get_selected_piece()
29         ↪ == other._state.get_selected_piece()
30
31     def child_already_exists(self, new_state: Quarto):
32         board_new_state = new_state.state_as_array()
33         for child in self._children:
34             if BoardTransforms.compare_boards(board_new_state,
35             ↪ child._state.state_as_array()):

```

```

31         return True
32
33     return False
34
35 def update(self, reward: int):
36     self.visits += 1
37     self.wins += reward
38
39 def reward(self, player_id):
40     player_last_moved = 1 - self._state.get_current_player()
41
42     player_who_last_moved = 1 - self._state.get_current_player()
43
44     # 0 if plays first, 1 if plays second
45     agent_position = player_id
46
47     if player_who_last_moved == agent_position and 1 -
48         ↪ self._state.check_winner() == agent_position:
49         # MCTS won
50         return 1
51     elif player_who_last_moved == 1 - agent_position and 1 -
52         ↪ self._state.check_winner() == 1 - agent_position:
53         # MCTS lost
54         return 0
55     elif self._state.check_winner() == -1:
56         # Draw game
57         return 0.5
58
59 def find_random_child(self):
60     free_positions = []
61     board = self._state.state_as_array()
62     for i in range(4):
63         for j in range(4):
64             if board[i][j] == -1:
65                 free_positions.append((i, j))
66     place = random.choice(free_positions)
67     new_quarto = deepcopy(self._state)
68     # new_quarto = Quarto(board=self._state.state_as_array(),
69     ↪ selected_piece=self._state.get_selected_piece(),
70     ↪ curr_player=self._state.get_current_player())
71     new_quarto.place(place[1], place[0])
72     if new_quarto.check_finished() or new_quarto.check_winner() != -1:
73         final_point = True
74     else:
75         new_board =
76         ↪ list(itertools.chain.from_iterable(new_quarto.state_as_array()))
77     free_pieces = [piece for piece in range(0, 16) if piece not in
78         ↪ new_board]
79     piece = random.choice(free_pieces)
80     new_quarto.select(piece)

```

```

75         final_point = False
76         new_quarto._current_player = 1 - new_quarto._current_player
77         return Node(new_quarto, place, final_point)

```

```

1  '''
2  In this file, we build an MCTS player using a different, simpler node structure.
3  '''
4
5  import copy
6  import itertools
7  import logging
8  import math
9  import random
10 from lib.players import Player
11 from quarto.objects import Quarto
12 from .node import Node
13
14 class MCTS(Player):
15     def __init__(self, board, player_id = 0):
16         '''
17         Initialise player with empty children dictionary
18         and player id (indicates position MCTS plays)
19         This is important for reward function.
20         '''
21         # by default MCTS is player 0
22         self.children = dict()
23         self._player_id = player_id
24         super().__init__(board)
25
26     def uct(self, node, child):
27         '''
28         Apply UCT formula to select best child
29         Formula:  $UCT = wins/visits + \sqrt{2 \cdot \log(parent\_visits) / child\_visits}$ 
30         '''
31         return child._wins/child._visits +
32             ↪ math.sqrt(2*math.log(node._visits)/child._visits)
33
34     def select(self, node: Node):
35         '''
36         Select the child with the highest UCT value
37         '''
38         points = []
39         for child in self.children[node]:
40             points.append((child, self.uct(node, child)))
41
42         return max(points, key=lambda x: x[1])[0]
43
44     def traverse(self, node: Node):
45         '''

```

```

45     Traverse the tree to find the leaf node
46     '''
47     path = []
48     while True:
49         path.append(node)
50         if node not in self.children or not not self.children[node]:
51             return path
52
53         unexplored = self.children[node] - self.children.keys()
54         if unexplored:
55             path.append(unexplored.pop())
56             return path
57         node = self.select(node)
58
59     def expand(self, node: Node):
60         '''
61         Expands from the leaf node to a state that is hopefully terminal. In this
62         ↪ approach (different from MCTS1), the next piece is not passed down to the
63         ↪ next node, but is directly applied to all empty positions.
64         '''
65         if node.final_point:
66             self.children[node] = None
67             return
68
69         free_places = []
70         board = node._state.state_as_array()
71         for i in range(4):
72             for j in range(4):
73                 if board[i][j] == -1:
74                     free_places.append((i, j))
75
76         children = []
77         for y, x in free_places:
78             quarto = copy.deepcopy(node._state)
79             quarto.place(x, y)
80             if quarto.check_finished() or quarto.check_winner() != -1:
81                 n = Node(copy.deepcopy(quarto), (x, y), True)
82                 children.append(n)
83             else:
84                 free_pieces = [i for i in range(16) if i not in list(
85                     itertools.chain.from_iterable(quarto.state_as_array()))]
86                 for piece in free_pieces:
87                     new_quarto = copy.deepcopy(quarto)
88                     new_quarto.select(piece)
89                     new_quarto._current_player = (
90                         new_quarto._current_player + 1) % 2
91                     child = Node(new_quarto, (x, y))
92                     children.append(child)
93         self.children[node] = children

```

```

93     def simulate(self, node: Node):
94         """
95         Simulate until terminal state is reached
96         """
97         while True:
98             if node.final_point:
99                 reward = node.reward(self._player_id)
100                 return reward
101                 node = node.find_random_child()
102
103     def backpropagate(self, reward, path):
104         """
105         Backpropagate reward to all nodes in path
106         (Invert rewards based on player id)
107         """
108         for node in reversed(path):
109             node.update(reward)
110             reward = 1 - reward
111
112     def best_child(self, node: Node):
113         """
114         Choose best child purely based on wins and visits
115         """
116         if node.final_point:
117             raise RuntimeError(f'called on unterminal node')
118
119         def score(n):
120             logging.debug(f"Before reading in choose {n}")
121             if n.visits == 0:
122                 return float('-inf')
123             return self.wins[n] / self.visits[n]
124
125         return max(self.children[node], key=score)
126
127     def search(self, node: Node):
128         """
129         1. Traverse tree to find leaf node
130         2. Expand leaf node
131         3. Simulate from leaf node until terminal state is reached
132         4. Backpropagate reward to all nodes in path
133         """
134         path = self.traverse(node)
135         leaf = path[-1]
136         self.expand(leaf)
137         reward = self.simulate(leaf)
138         self.backpropagate(reward, path)
139
140     def do_rollout(self, root: Quarto):
141         """
142         Create node and rollout from it

```

```

143     '''
144     if type(root) != Node:
145         root = Node(state=root)
146     self.search(root)
147     return self.best_child(root)
148
149     def choose_piece(self):
150         '''
151         Subclassed from Calabrese's player class. Will return a random piece if
152         ↪ first move. If not, will return piece computed in `place_piece`
153         '''
154         if self.mcts_last_board == None:
155             return random.randint(0, 15)
156         else:
157             return self.mcts_last_board._state.get_selected_piece()
158
159     def place_piece(self):
160         '''
161         Iterate through and rollout before returning best child (next move to
162         ↪ make)
163         Since parent player class expects position and next piece to be
164         returned by separate functions, next piece is stored in a variable in
165         ↪ order to be called by `choose_piece`
166         '''
167         board = self.get_game().state_as_array()
168         selected_piece = self.get_game().get_selected_piece()
169         curr_player = self.get_game().get_current_player()
170         current_board = Quarto(
171             board=board, selected_piece=selected_piece, curr_player=curr_player)
172         root = Node(current_board)
173         self._player_id = self.get_game().get_current_player()
174         for _ in range(30):
175             best_child = self.do_rollout(root)
176         self.mcts_last_board = best_child
177         return best_child.place_current_move

```

5.4.3 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):
44         '''
45         Returns a piece to be placed on the board
46         '''
47         state = self.get_game()
48
49         possible_pieces = []
50         for i in range(16):
51             # check if the piece is a winning piece
52             winning_piece, _ = self.check_if_winning_piece(state, i)
53             if (not winning_piece) and (i not in
54                 ↪ list(itertools.chain.from_iterable(state.state_as_array())) and
55                 ↪ (i != state.get_selected_piece())):
56                 possible_pieces.append(i)
57
58         # if no pieces can be placed on board anymore (board full/game over),
59         ↪ return -1

```

```

57     if len(possible_pieces) == 0:
58         # check if number of non-empty cells is 16
59         if len([i for i in
        ↪ list(itertools.chain.from_iterable(state.state_as_array())) if i
        ↪ != -1]) == 16:
60             return -1
61         else:
62             # there are possible pieces to be placed, but they are winning
        ↪ pieces/already in board
63             on_board = list(itertools.chain.from_iterable(
64                 state.state_as_array()))
65             not_on_board = list(set(range(16)) - set(on_board))
66             return random.choice(not_on_board)
67     else:
68         return random.choice(possible_pieces)
69
70     def choose_piece(self):
71         '''
72         Returns a piece to be placed on the board
73         '''
74         return self.hardcoded_strategy_get_piece()
75
76     def hardcoded_strategy_get_move(self, return_winning_piece_boolean=True):
77         # 1. Play the piece handed over by the opponent:
78         # (a) play a winning position if handed a winning piece;
79         # (b) otherwise, play to build a line of like pieces if possible;
80         # (c) otherwise, play randomly.
81         # 2. Hand a piece to the opponent:
82         # (a) avoid handing over a winning piece for your opponent to play;
83         # (b) otherwise, choose randomly.
84
85         state = self.get_game()
86
87         board = state.state_as_array()
88         selected_piece = state.get_selected_piece()
89         # check if the selected piece is a winning piece
90         winning_piece, position = self.check_if_winning_piece(
91             state, selected_piece)
92         if winning_piece:
93             return selected_piece, position
94
95         # check if the selected piece can be used to build a line of like pieces
96
97         row_1 = [[0, 0], [0, 1], [0, 2], [0, 3]]
98         # pieces in row 2
99         row_2 = [[1, 0], [1, 1], [1, 2], [1, 3]]
100        # pieces in row 3
101        row_3 = [[2, 0], [2, 1], [2, 2], [2, 3]]
102        # pieces in row 4
103        row_4 = [[3, 0], [3, 1], [3, 2], [3, 3]]

```



```

104
105     # pieces in column 1
106     col_1 = [[0, 0], [1, 0], [2, 0], [3, 0]]
107     # pieces in column 2
108     col_2 = [[0, 1], [1, 1], [2, 1], [3, 1]]
109     # pieces in column 3
110     col_3 = [[0, 2], [1, 2], [2, 2], [3, 2]]
111     # pieces in column 4
112     col_4 = [[0, 3], [1, 3], [2, 3], [3, 3]]
113
114     # pieces in diagonal 1
115     diag_1 = [[0, 0], [1, 1], [2, 2], [3, 3]]
116     # pieces in diagonal 2
117     diag_2 = [[0, 3], [1, 2], [2, 1], [3, 0]]
118
119     for line in [row_1, row_2, row_3, row_4, col_1, col_2, col_3, col_4,
120                 ↪ diag_1, diag_2]:
121         # check if the selected piece can be used to build a line of like
122         ↪ pieces
123         characteristics = []
124         empty_rows = []
125         for el in line:
126             x, y = el
127             if board[x, y] != -1:
128                 piece = board[x][y]
129                 piece_char = state.get_piece_characteristics(piece)
130                 characteristics.append(
131                     [piece_char.HIGH, piece_char.COLOURED, piece_char.SOLID,
132                     ↪ piece_char.SQUARE])
133             else:
134                 empty_rows.append(el)
135                 characteristics.append([-1, -1, -1, -1])
136
137     selected_piece_char = state.get_piece_characteristics(
138         selected_piece)
139     selected_piece_char = [selected_piece_char.HIGH,
140     ↪ selected_piece_char.COLOURED,
141                             selected_piece_char.SOLID,
142     ↪ selected_piece_char.SQUARE]
143
144     # check if characteristics has an empty row
145     if [-1, -1, -1, -1] in characteristics:
146         # count how many [-1, -1, -1, -1] are in characteristics
147         empty_indexes = [i for i, x in enumerate(

```

```

148     # proceeding to check couplets and see if they can build
149     ↪ triplets
150     # since 2 empty rows may be present and either could create a
151     ↪ triplet, have to choose randomly later
152     potential_moves = []
153
154     for i, index in enumerate(empty_indexes):
155         position = empty_rows[i]
156         # insert the selected piece in the empty row
157         # empty_piece_index = characteristics.index(
158         #     [-1, -1, -1, -1])
159         characteristics = characteristics_copy.copy()
160         characteristics[index] = selected_piece_char
161
162         # check if any column has the same characteristics
163         col1 = [characteristics[0][0], characteristics[1][0],
164                 characteristics[2][0], characteristics[3][0]]
165         col2 = [characteristics[0][1], characteristics[1][1],
166                 characteristics[2][1], characteristics[3][1]]
167         col3 = [characteristics[0][2], characteristics[1][2],
168                 characteristics[2][2], characteristics[3][2]]
169         col4 = [characteristics[0][3], characteristics[1][3],
170                 characteristics[2][3], characteristics[3][3]]
171
172         col1 = [int(i) for i in col1]
173         col2 = [int(i) for i in col2]
174         col3 = [int(i) for i in col3]
175         col4 = [int(i) for i in col4]
176
177         # print(col1, col2, col3, col4)
178         def check_if_form_triplet(line):
179             # earlier we checked if we can complete a line
180             # here we check if we can form a triplet (one step away
181             ↪ from completing a line)
182             return line.count(1) == 3 or line.count(0) == 3
183
184         # if len(set(col1)) == 1 or len(set(col2)) == 1 or
185         ↪ len(set(col3)) == 1 or len(set(col4)) == 1:
186         if check_if_form_triplet(col1) or check_if_form_triplet(col2)
187         ↪ or check_if_form_triplet(col3) or
188         ↪ check_if_form_triplet(col4):
189             # this piece can be used to build a line of like pieces
190             logging.debug('playing to build a line of like pieces')
191             potential_moves.append(list(reversed(position)))
192
193     if len(potential_moves) >= 1:
194         if return_winning_piece_boolean:
195             # return True, list(reversed(empty_rows[-1]))
196             return True, random.choice(potential_moves)
197         else:

```

```

192         # move = list(reversed(empty_rows[-1]))
193         # move = list(reversed(position))
194         move = random.choice(potential_moves)
195         return move[0], move[1]
196
197     # play randomly
198     possible_moves = []
199     for i in range(self.BOARD_SIDE):
200         for j in range(self.BOARD_SIDE):
201             for next_piece in range(16):
202                 if state.check_if_move_valid(selected_piece, i, j,
203                 ↪ next_piece):
204                     if return_winning_piece_boolean:
205                         possible_moves.append([False, [i, j]])
206                     else:
207                         possible_moves.append([i, j])
208
209     random_move = random.choice(possible_moves)
210     return random_move[0], random_move[1]
211
212     logging.debug(f"Selected piece: {selected_piece}")
213     logging.debug(f"Board: {board}")
214     logging.debug('no move found')
215
216     def place_piece(self):
217         '''
218         Above function sometimes necessary to return additional information
219         In game, first return value is not necessary
220         '''
221         return
222         ↪ self.hardcoded_strategy_get_move(return_winning_piece_boolean=False)

```

5.4.4 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.

```

1  import sys
2  sys.path.insert(0, '..')
3
4  from collections import defaultdict
5  from copy import deepcopy
6  import itertools
7  import json
8  import logging
9  import math
10 import os
11 import random

```

```

12 import time
13
14 # from MCTS import MonteCarloTreeSearch
15 from MCTS.mcts import decode_tree
16 from MCTS2.mcts import MCTS
17 from quarto.objects import Quarto
18 from lib.players import Player, RandomPlayer
19 from lib.isomorphic import BoardTransforms
20
21 import tqdm
22 logging.basicConfig(level=logging.DEBUG)
23
24
25 class QLearningPlayer(Player):
26     def __init__(self, board: Quarto = Quarto(), epsilon=0.1, alpha=0.5,
27         ↪ gamma=0.9, tree: MCTS = None):
28         self.epsilon = epsilon
29         self.alpha = alpha
30         self.gamma = gamma
31         self.board = board
32         self.MAX_PIECES = 16
33         self.BOARD_SIDE = 4
34         self.Q = defaultdict(int)
35
36         if tree is not None:
37             # load the pre-initialised tree
38             self.tree = tree
39             self.tree.set_board(board)
40
41         else:
42             # load new tree
43             self.tree = MCTS(board=board)
44
45         super().__init__(board)
46
47     def clear_and_set_current_state(self, state: Quarto):
48         self.current_state = state
49         self.tree = MCTS(board=state)
50
51     def reduce_normal_form(self, state: Quarto):
52         '''
53         Reduce the Quarto board to normal form (i.e. the board is symmetric)
54         '''
55         # NOT IMPLEMENTED for now, just return the board
56         return state
57
58     def hash_state_action(self, state: Quarto, action):
59         # reduce to normal form before saving to Q table
60         return state.board_to_string() + '||' + str(state.get_selected_piece()) +
61         ↪ '||' + str(action)

```

```

60
61 def get_Q(self, state, action):
62     # check possible transforms first (really really slow)
63     for key, val in self.Q.items():
64         if BoardTransforms.compare_boards(state.state_as_array(),
65         ↪ state.string_to_board(key.split('||')[0])):
66             return val
67
68     if self.hash_state_action(state, action) not in self.Q:
69         # used to determine if state exists in Q table
70         # if None, then go to MCTS
71         return None
72
73     return self.Q[self.hash_state_action(state, action)]
74
75 def get_Q_for_state(self, state):
76     if self.hash_state_action(state, None) not in self.Q:
77         return None
78     return [i for i in self.Q if i.startswith(str(state))]
79
80 def set_Q(self, state, action, value):
81     self.Q[self.hash_state_action(state, action)] = value
82
83 def get_possible_actions(self, state: Quarto):
84     actions = []
85     for i in range(self.BOARD_SIDE):
86         for j in range(self.BOARD_SIDE):
87             for piece in range(self.MAX_PIECES):
88                 if state.check_if_move_valid(self.board.get_selected_piece(),
89                 ↪ i, j, piece):
90                     actions.append((i, j, piece))
91
92     return actions
93
94 def get_max_Q(self, state):
95     max_Q = -math.inf
96     for action in self.get_possible_actions(state):
97         if self.get_Q(state, action) is not None:
98             Q_val = self.get_Q(state, action)
99             max_Q = max(max_Q, self.get_Q(state, action))
100     return max_Q
101
102 def get_action(self, state, mode='testing'):
103     '''
104     If state, action pair not in Q, go to Monte Carlo Tree Search to find
105     ↪ best action
106     '''
107     if mode == 'training':
108         # exploration through epsilon greedy
109         # look for good moves through Monte Carlo Tree Search

```

```

1107     if random.random() < self.epsilon:
1108         # for i in range(10):
1109         #     self.tree.do_rollout(state)
1110         best_action = self.tree.place_piece()
1111         return best_action
1112     else:
1113         # look in the q table for the best action
1114         expected_score = 0
1115         best_action = None
1116         for action in self.get_possible_actions(state):
1117             if self.get_Q(state, action) is not None and expected_score <
1118                 ↪ self.get_Q(state, action):
1119                 print('found in Q table')
1120                 expected_score = self.get_Q(state, action)
1121                 best_action = action
1122         # go to Monte Carlo Tree Search if no suitable action found in Q
1123         ↪ table
1124         if best_action is None or expected_score == 0:
1125             logging.debug(
1126                 'No suitable action found in Q table, going to Monte
1127                 ↪ Carlo Tree Search')
1128             for i in range(10):
1129                 self.tree.do_rollout(state)
1130             best_action = self.tree.place_piece()
1131         else:
1132             print('found in Q table')
1133
1134         return best_action
1135     else:
1136         # in test mode, use the Q table to find the best action
1137         # only go to Monte Carlo Tree Search if no suitable action found in Q
1138         ↪ table
1139         expected_score = 0
1140         best_action = None
1141         for action in self.get_possible_actions(state):
1142             if self.get_Q(state, action) is not None and expected_score <
1143                 ↪ self.get_Q(state, action):
1144                 expected_score = self.get_Q(state, action)
1145                 best_action = action
1146         # go to Monte Carlo Tree Search if no suitable action found in Q
1147         ↪ table
1148         if best_action is None or expected_score == 0:
1149             logging.debug(
1150                 'No suitable action found in Q table, going to Monte Carlo
1151                 ↪ Tree Search')
1152             # for i in range(20):
1153             #     print('doing rollout')
1154             #     self.tree.do_rollout(state)
1155             best_action = self.tree.place_piece()
1156         return best_action

```

```

150
151 def update_Q(self, state, action, reward, next_state):
152     Q_val = self.get_Q(state, action)
153     if Q_val is None:
154         Q_val = random.uniform(1.0, 0.01)
155     self.set_Q(state, action, Q_val + self.alpha *
156         (reward + self.gamma * self.get_max_Q(next_state) - Q_val))
157
158 def train(self, iterations=100):
159     # 1. Use the Q-function to initialize the value of each state-action
160     → pair,  $Q(s, a) = 0$ .
161     # automatically done through defaultdict
162
163     # Choose an action using MCTS
164     wins = 0
165     tries = 0
166     agent_decision_times = []
167
168     progress_bar = tqdm.tqdm(total=iterations)
169     for i in range(iterations):
170         board = Quarto()
171         self.board = board
172         random_player = RandomPlayer(board)
173         self.tree.set_board(board)
174         self.current_state = board
175         self.previous_state = None
176         self.previous_action = None
177         player = 1
178         self.current_state.switch_player()
179         selected_piece = random_player.choose_piece()
180         self.current_state.set_selected_piece(selected_piece)
181         while True:
182             reward = 0
183             if player == 0:
184                 # QL-MCTS moves here
185                 print('QL-MCTS moves here')
186                 self.previous_state = deepcopy(self.current_state)
187                 logging.debug("Piece to place: ",
188                     self.current_state.get_selected_piece())
189                 logging.debug("Board: ")
190                 logging.debug(self.current_state.state_as_array())
191                 time_start = time.time()
192                 action = self.get_action(self.current_state)
193                 next_piece = self.tree.choose_piece()
194                 self.previous_action = (action[0], action[1], next_piece)
195                 time_end = time.time()
196                 agent_decision_times.append(time_end - time_start)
197                 self.current_state.select(selected_piece)
198                 self.current_state.place(action[0], action[1])
199                 self.current_state.set_selected_piece(next_piece)

```

```

200         self.current_state.switch_player()
201         player = 1 - player
202     else:
203         # Random moves here
204         action = random_player.place_piece()
205         next_piece = random_player.choose_piece()
206         while
207             ↪ self.board.check_if_move_valid(self.board.get_selected_piece(),
208             ↪ action[0], action[1], next_piece) is False:
209                 action = random_player.place_piece()
210                 next_piece = random_player.choose_piece()
211         self.current_state.select(
212             self.current_state.get_selected_piece())
213         self.current_state.place(action[0], action[1])
214         self.current_state.set_selected_piece(next_piece)
215         self.current_state.switch_player()
216         player = 1 - player
217
218     if self.current_state.check_is_game_over():
219         if 1 - self.current_state.check_winner() == 1:
220             logging.info('QL-MCTS won')
221             reward = 1
222             wins += 1
223         else:
224             logging.info('Random won')
225             reward = -1
226         self.update_Q(self.previous_state, self.previous_action,
227                       reward, self.current_state)
228         break
229     else:
230         if self.previous_state is not None:
231             self.update_Q(
232                 self.previous_state, self.previous_action, reward,
233                 ↪ self.current_state)
234
235     tries += 1
236     if i % 10 == 0:
237         logging.info(f'Iteration {i}')
238         logging.info(f'Wins: {wins}')
239         logging.info(f'Tries: {tries}')
240         logging.info(f'Win rate: {wins/tries}')
241         wins = 0
242         tries = 0
243
244     # OPTION 1: clear the tree every time
245     self.tree = MCTS(board=self.board)
246
247     # OPTION 2: if average agent decision time is too long, clear the
248     ↪ MCTS tree

```



```

245         # if sum(agent_decision_times) / len(agent_decision_times) > 5:
246         #     self.tree = MonteCarloTreeSearch(board=self.board)
247         #     agent_decision_times = []
248
249         progress_bar.update(1)
250
251
252 if __name__ == '__main__':
253     # load tree with MonteCarloSearchDecoder
254     # with open('progress.json', 'r') as f:
255     #     tree = decode_tree(json.load(f))
256     qplayer = QLearningPlayer()
257     qplayer.train(10)

```

5.5 Utility Functions

5.5.1 OpenAI Gym Environment for Quarto

Though the DQN is abandoned, I leave this here for posterity.

```

1 class QuartoScapeNew(gym.Env):
2     '''Custom gym environment for Quarto'''
3     def __init__(self):
4         self.game = Quarto()
5         self.action_space = spaces.MultiDiscrete([16, 16, 16])
6         self.observation_space = spaces.MultiDiscrete([17] * 17)
7         self.reward_range = (-1, 1)
8         self.main_player = None
9
10    def set_main_player(self, player):
11        self.main_player = player
12        self.game.set_players((player, RandomPlayer(self.game)))
13        return True
14
15    def step(self, action, chosen_piece):
16        # position is the position the previous piece should be moved to
17        # chosen next piece is the piece the agent chooses for the next player to
18        #   ↪ move
19        x, y, chosen_next_piece = action
20        self.next_piece = chosen_next_piece
21        if self.game.check_if_move_valid(chosen_piece, x, y, chosen_next_piece):
22            # print(f"Valid move, piece {chosen_piece} placed at {x}, {y}")
23            self.game.select(chosen_piece)
24            self.game.place(x, y)
25            # self.game.print()
26            if self.game.check_is_game_over():
27                # just playing with itself
28                logging.info("Giving reward of 1 for completing the game")
29                reward = 1

```

```

29         return self.game.state_as_array(), self.game.check_winner(),
        ↪ self.game.check_finished(), {}
30     else:
31         logging.info("Giving reward of 0 for making a move that didn't
        ↪ end the game")
32         reward = 0
33         return self.game.state_as_array(), self.game.check_winner(),
        ↪ self.game.check_finished(), {}
34
35     else:
36         reward = -1
37
38         return self.game.state_as_array(), reward, self.game.check_finished(), {}
39
40     def reset(self):
41         self.game = Quarto()
42         self.game.set_players((self.main_player, RandomPlayer(self.game)))
43         # print(self.game.state_as_array())
44         return self.game.state_as_array()

```

5.6 Code for Unsuccessful Players

5.6.1 Slower MCTS With Different Node Structure

The implementation of MCTS and the rollout strategy is based on the minimal implementation [here](#). It is slower but performs better than the MCTS strategy in the previous section due to a higher expansion factor, since it also takes into account every possible next piece that can be chosen for the next player when finding children.

```

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     '''
45     Recreate MonteCarloTreeSearch object from JSON
46     '''
47
48     def __init__(self, *args, **kwargs):
49         json.JSONDecoder.__init__(
50             self, object_hook=self.object_hook, *args, **kwargs)
51
52     def object_hook(self, obj):
53         children = {}
54
55         for k, v in obj['children'].items():
56             children[Node(hash_state=k)] = [
57                 NodeDecoder().object_hook(node) for node in v]
58
59         if 'Q' in obj:
60             return MonteCarloTreeSearch(
61                 Q={Node(hash_state=k): v for k, v in obj['Q'].items()},
62                 N={Node(hash_state=k): v for k, v in obj['N'].items()},
63                 children=children,
64                 epsilon=obj['epsilon'],
65             )
66         return obj

```

```

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
94         self.BOARD_SIDE = 4
95         self.board = board
96         self.random_factor = 0
97         self.decisions = 0
98         super().__init__(board)
99
100     def set_board(self, board):
101         self.board = board
102
103     def choose(self, node):
104         '''
105         Choose best successor of node (move)
106         Returns the board itself
107         '''
108         def score(n):
109             logging.debug(f"Before reading in choose {n}")
110             if self.N[n] == 0:
111                 return float('-inf')
112             return self.Q[n] / self.N[n]
113
114         # node is board Quarto
115         node = Node(node)
116         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
143
144 def choose_piece(self):
145     '''
146     Choose a piece to make the opponent place
147     '''
148     node = Node(board=self.board,
149                 selected_piece_index=self.board.get_selected_piece())
150
151     if node.is_terminal():
152         logging.debug(node.board.state_as_array())
153         raise RuntimeError("choose called on terminal node")
154
155     if node not in self.children:
156         # index -1 of tuple is next piece from a board
157         print("Random child")
158         return node.find_random_child()[-1]
159
160     def score(n):
161         logging.debug(f"Before reading in choose {n}")
162         if self.N[n] == 0:
163             return float('-inf')
164         return self.Q[n] / self.N[n]

```

```

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
192             # number of times have to resort to random
193             rand_child = node.find_random_child()
194             print("Random child")
195             # add to children
196             return rand_child.board.move
197
198     def score(n):
199         logging.debug(f"Before reading in choose {n}")
200         if self.N[n] == 0:
201             return float('-inf')
202         return self.Q[n] / self.N[n]
203
204         # print("In place piece")
205         # print(max(self.children[node], key=score).move)
206         return max(self.children[node], key=score).move[1:]
207
208     def do_rollout(self, board):
209         """
210         Rollout from the node for one iteration
211         """
212         logging.debug("Rollout")
213         # if root node, there is no move
214         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)
241
242 def expand(self, node):
243     # logging.debug('Expanding')
244     if node in self.children:
245         return
246     self.children[node] = node.find_children()
247     # logging.debug('Children: ', self.children[node])
248
249 def simulate(self, node):
250     '''
251     Returns reward for random simulation
252     '''
253     invert_reward = False
254     while True:
255         if node.is_terminal():
256             reward = node.reward()
257
258             return 1 - reward if invert_reward else reward
259         node = node.find_random_child()
260         # invert_reward = not invert_reward
261
262 def backpropagate(self, path, reward):
263     '''

```

```

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
289         draws = 0
290         for i in range(num_trials):
291             board = Quarto()
292             random_player = RandomPlayer(board)
293             self.board = board
294             board.set_selected_piece(random_player.choose_piece(board))
295             while True:
296                 # random player moves
297                 chosen_location = random_player.place_piece(
298                     board, board.get_selected_piece())
299                 chosen_piece = random_player.choose_piece(board)
300                 while not board.check_if_move_valid(board.get_selected_piece(),
301                     ↪ chosen_location[0], chosen_location[1], chosen_piece):
302                     chosen_location = random_player.place_piece(
303                         board, board.get_selected_piece())
304                     chosen_piece = random_player.choose_piece(board)
305                 board.select(board.get_selected_piece())
306                 board.place(chosen_location[0], chosen_location[1])
307                 # setting the piece for the next player
308                 board.set_selected_piece(chosen_piece)
309                 board.switch_player()
310
311             if board.check_is_game_over():

```



```

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

```

```

358         # setting the piece for the next player
359         board.set_selected_piece(chosen_piece)
360         board.switch_player()
361
362         if board.check_is_game_over():
363             if 1 - board.check_winner() == 0:
364                 logging.info("Random player won")
365             else:
366                 logging.info("Draw")
367             break
368         # monte carlo tree search moves
369
370         # make move with monte carlo tree search
371         for _ in range(20):
372             self.do_rollout(board)
373         board = self.choose(board)
374
375         if board.check_is_game_over():
376             # TODO: check if it's a draw
377             if 1 - board.check_winner() == 1:
378                 logging.info("Agent won")
379             else:
380                 logging.info("Draw")
381             break
382         # don't need to switch player because it's done in choose
383         # random_player needs to do it because it is not done
384         ↪ automatically
385
386         if i % 2 == 0:
387             # run a test to see if the agent is improving
388             self.test_win_rate()
389
390         # save progress every 10 iterations
391         if i % 100 == 0:
392             logging.debug("Saving progress")
393             if save_format == 'json':
394                 self.save_progress_json('/Volumes/USB/progress3.json')
395             else:
396                 self.save_progress_pickle('progress.pkl')
397
398     def train(self):
399         '''
400         Train without multithreading
401         '''
402         self.train_engine(Quarto(), 100, 'json')
403
404     def threaded_training(self, num_threads=1, save_format='json'):
405         '''
406         Train the model
407         '''

```

```

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         # 1 is the default value, but it can be changed to 0.5 or 0.1
425
426         self.tau = 0.5
427         if node not in self.children:
428             self.do_rollout(root.board)
429
430         probs = [self.N[child] / self.N[root]
431                  for child in self.children[node]]
432
433         probs = [p ** (1 / self.tau) for p in probs]
434
435         probs = [p / sum(probs) for p in probs]
436
437         return probs
438
439     def save_progress_pickle(self, filename):
440         with open(filename, 'wb') as f:
441             pickle.dump(self, f)
442
443     def save_progress_json(self, filename):
444         with open(filename, 'w') as f:
445             json.dump(self, f, cls=MonteCarloTreeSearchEncoder)
446
447     def load_progress_json(self, filename):
448         with open(filename, 'r') as f:
449             return json.load(f, cls=MonteCarloTreeSearchDecoder)
450
451     def load_progress(self, filename):
452         with open(filename, 'rb') as f:
453             return pickle.load(f)
454
455
456 if __name__ == "__main__":

```

```

457 mcts = MonteCarloTreeSearch()
458 # with open('/Volumes/USB/progress3.json', 'r') as f:
459 #     mcts = decode_tree(json.load(f))
460 #     logging.info("Loaded progress")
461 logging.info("Starting training")
462 mcts.train()

```

5.6.2 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
83         ↪ is an action)
84         An action is made up of (x, y, piece chosen for next player)
85         There are 16 * 16 * 16 possible actions and the mapping is found in
86         ↪ get_all_actions()
87         '''
88         model = Sequential()
89         initializer = HeUniform()

```

```

88     model.add(Dense(
89         12, input_dim=self.env.observation_space.shape[0], activation='relu',
90         ↪ kernel_initializer=initializer))
91     model.add(Dense(24, activation='relu', kernel_initializer=initializer))
92     model.add(Dense(48, activation='relu', kernel_initializer=initializer))
93     model.add(Dense(96, activation='relu', kernel_initializer=initializer))
94     model.add(Dense(192, activation='relu',
95                     kernel_initializer=initializer))
96     model.add(Dense(4 * 4 * 16, activation='linear',
97                     kernel_initializer=initializer))
98     model.compile(loss=tf.keras.losses.Huber(), metrics=[
99         'mae', 'mse'], optimizer=Adam(learning_rate=0.001))
100
101     return model
102
103 def build_conv_model(self):
104     model = Sequential()
105     model.add(Conv2D(32, (3, 3), input_shape=(4, 4, 4), activation='relu'))
106     model.add(MaxPooling2D(pool_size=(2, 2)))
107     model.add(Flatten())
108     model.add(Dense(16, activation='relu'))
109     model.add(Dense(4 * 4 * 16, activation='linear'))
110     model.compile(loss='mse', metrics=[
111         'accuracy'], optimizer=Adam(learning_rate=0.001))
112
113     return model
114
115 def get_position(self, element, list):
116     if element in list:
117         return list.index(element)
118     else:
119         return -1
120
121 def make_prediction(self, state, chosen_piece=None):
122     '''Make a prediction using the network'''
123     # prediction X is the position of the single 1 in the state
124     pred_X = [self.get_position(i, list(state.flatten()))
125               for i in range(0, 16)]
126     pred_X.append(chosen_piece)
127     return self.model.predict(np.array([pred_X]), verbose=0)[0]
128
129 def decay_lr(self, lr, decay_rate, decay_step):
130     return lr * (1 / (1 + decay_rate * decay_step))
131
132 def abbellire(self, state, chosen_piece):
133     '''
134         Beautify the state for network input
135         When in Italy, do as the Italians do
136     '''
137     X = [self.get_position(i, list(state.flatten())) for i in range(0, 16)]
138     X.append(chosen_piece)
139     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.abellire(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.abellire(new_current_state,
160     ↪ action[2])
161                                   ↪ for state, chosen_piece, action, reward,
162                                   ↪ new_current_state, done in
163                                   ↪ minibatch])
164
165     future_qs = self.target_model.predict(
166         new_current_states.reshape(batch_size, 17), verbose=0)
167     # exclude invalid moves from calculation
168     X = []
169     Y = []
170     for index, (current_state, chosen_piece, action, reward,
171     ↪ new_current_state, done) in enumerate(minibatch):
172         if not done:
173             # max_future_q = np.max(future_qs[index])
174             # new_q = reward + self.gamma * max_future_q
175             max_future_q = reward + self.gamma * np.max(future_qs[index])
176         else:
177             # max_future_q = reward
178             max_future_q = reward
179
180     # 0 2 5
181     # 0 + 2 * 4 + 5 * 16 = 85
182     current_qs[index][action[0] + action[1] * 4 + action[2] * 16] = (
183         1 - self.lr) * current_qs[index][action[0] + action[1] * 4 +
184         ↪ action[2] * 16] + self.lr * max_future_q
185
186     X.append(self.abellire(current_state, chosen_piece))
187     Y.append(current_qs[index])

```

```

179     X = np.array(X).reshape(batch_size, 17)
180     Y = np.array(Y).reshape(batch_size, 4 * 4 * 16)
181     logging.debug(X)
182     logging.debug(Y)
183     self.model.fit(X, Y, batch_size=batch_size,
184                   verbose=1, shuffle=True, epochs=1)
185
186
187 def choose_piece(self, state: Any, piece_chosen_for_you: int):
188     '''Choose piece for the next guy to play'''
189     self.env.game.set_board(state)
190     pred = self.make_prediction(state, piece_chosen_for_you)
191     pred = self.nan_out_invalid_actions(-100, pred)
192     best_action = np.nanargmax(pred)
193     best_action = self.get_all_actions()[best_action]
194     return best_action[2]
195
196 def place_piece(self, state: Any, piece_chosen_for_you: int):
197     '''Choose position to move piece to based on the current state'''
198     self.env.game.set_board(state)
199     pred = self.make_prediction(state, piece_chosen_for_you)
200     pred = self.nan_out_invalid_actions(piece_chosen_for_you, pred)
201     best_action = np.nanargmax(pred)
202     best_action = self.get_all_actions()[best_action]
203     # print(f'Best action for place piece: {best_action}')
204     return best_action[0], best_action[1]
205
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

```



```

227 num_episodes = 2000
228
229 steps_to_update_target_model = 0
230
231 for episode in range(num_episodes):
232     if episode % 100 == 0:
233         self.model.save(f'/Volumes/USB/qn_weights.h5')
234
235     total_training_reward = 0
236     print(f'Episode: {episode}')
237     state = self.env.reset()
238     done = False
239     # initialise chosen piece with a random piece
240     # in reality, the opponent will choose a piece for you
241     chosen_piece = random.randint(0, 15)
242     while not done:
243         steps_to_update_target_model += 1
244
245         if random.random() < self.epsilon:
246             action = self.env.action_space.sample()
247             while not self.env.game.check_if_move_valid(chosen_piece,
248                 ↪ action[0], action[1], action[2]):
249                 action = self.env.action_space.sample()
250         else:
251             prediction = self.make_prediction(state, chosen_piece)
252             prediction = self.nan_out_invalid_actions(
253                 chosen_piece, prediction)
254             if np.all(np.isnan(prediction)):
255                 action = self.env.action_space.sample()
256                 while not self.env.game.check_if_move_valid(chosen_piece,
257                     ↪ action[0], action[1], action[2]):
258                     action = self.env.action_space.sample()
259             else:
260                 action = np.nanargmax(prediction)
261                 # get action at index of action
262                 action = self.get_all_actions()[action]
263
264     new_state, reward, done, _ = self.env.step(
265         action, chosen_piece)
266
267     replay_memory.append(
268         (state, chosen_piece, action, reward, new_state, done))
269
270     if done:
271         logging.debug('GAME OVER')
272
273     if steps_to_update_target_model % 4 == 0 or done:
274         self.train(replay_memory, 32)
275
276     state = new_state

```

```

275         total_training_reward += reward
276
277         if done:
278             total_training_reward += 1
279
280             if steps_to_update_target_model >= 100:
281                 self.target_model.set_weights(self.model.get_weights())
282                 steps_to_update_target_model = 0
283             break
284
285         chosen_piece = action[2]
286
287         if episode % 10 == 0:
288             logging.info(f'Testing win rate after {episode} episodes')
289             test(self)
290
291             self.lr = self.decay_lr(self.lr, 0.0001, episode)
292
293         self.env.close()
294         self.model.save('/Volumes/USB/qn_weights.h5')
295
296 def main():
297     dq_wins = 0
298     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()

```

6 Conclusion and Final Considerations

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

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