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 Received Reviews . . . 13
2.4 Given Reviews . . . 15
2.4.1 Shayan . . . 15
2.4.2 Arman . . . 16

3 Lab 2 **19**
3.1 Solution . . . 19
3.1.1 Representing the problem . . . 19
3.1.2 Assessing Fitness . . . 19
3.2 Results . . . 20
3.2.1 The Case of Mutations . . . 20
3.3 Mutation Functions . . . 21
3.3.1 Flip Mutation . . . 21
3.3.2 Scramble Mutation . . . 22
3.3.3 Swap Mutation . . . 22
3.3.4 Inversion Mutation . . . 22
3.4 Full Code . . . 23
3.5 Received Reviews . . . 29
3.6 Given Reviews . . . 31
3.6.1 Erik . . . 31
3.6.2 Karl . . . 35
3.6.3 Ricardo . . . 40
3.6.4 Francesco . . . 45

4 Lab 3 **49**
4.1 Solution . . . 49
4.1.1 Fixed Rules . . . 49
4.1.2 Evolved Agent Approach 1 . . . 55
4.1.3 Evolved Agent Approach 2 (Probability Thresholds) . . . 64
4.1.4 Minmax . . . 67
4.1.5 Reinforcement Learning . . . 70
4.2 Acknowledgements . . . 78
4.3 Received Reviews . . . 78
4.4 Given Reviews . . . 80
4.4.1 Karl . . . 80

5 Final Project **94**
5.1 My Strategy For Solving the Problem . . . 94
5.1.1 Step 1: Implement and Tune Multiple Search Algorithms . . . 94
5.1.2 Step 2: Analysing the Algorithms . . . 95
5.1.3 Step 3: Implementing the Hybrid Agent . . . 96
5.2 Code . . . 97
5.3 Acknowledgements . . . 97

6 Conclusion **99**

1 Introduction

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

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

2 Lab 1

2.1 Solution

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

1. Naive Greedy
2. Greedy with a better cost function
3. A* Traversal Using a Priority Queue
4. A* Traversal Using a Fully Connected Graph

2.1.1 Naive Greedy

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

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

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

2.1.2 Greedy with basic heuristic approximation

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

the current subset:

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

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

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

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

2.1.3 A* Search Using a Priority Queue

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

The implemented algorithm can be surmised as pseudocode below:

1. Add the start node to the priority queue
2. While the state is not None, cycle through the subsets and compute the cost of adding this subset to the final list.
3. If the cost has not been stored yet and the the new state is not in the queue, update the parent of each state. If travelling in this route produces a cheaper cost, update the cost of the node and its parent.
4. Finally, compute the path we travelled through.

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

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

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

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

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

    def h(self, state):
```

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

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

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

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

```
                    state = frontier.pop()
                    visited_nodes += 1
                else:
                    state = None

            path = list()
            s = state

            while s:
                path.append(s.copy_data())
                s = parents[s]

            print(f"Length of final list: {len(self.flatten_list(path[0]))}")
            print(f"Found a solution in {len(path):,} steps; visited
            ↪  {len(cost_of_each_state):,} states")
            print(f"Visited {visited_nodes} nodes")
            print(
                f"My solution for N={self.N}: w={sum(len(_) for _ in path[0])}
                ↪  (bloat={(sum(len(_) for _ in
                ↪  path[0])-self.N)/self.N*100:.0f}%)"
            )
            return list(reversed(path))

        def search(self, constant_cost=False):
            GOAL = State(np.array(range(self.N)))
            subsets = problem(self.N, seed=self.seed)
            initial_state = State(np.array([subsets[0]]))

            parents = dict()
            cost_of_each_state = dict()

            self.astar_search(
                initial_state = initial_state,
                subsets = subsets,
                parents = parents,
                cost_of_each_state = cost_of_each_state,
                priority_function = lambda state: cost_of_each_state[state] +
                ↪  self.h(state),
                unit_cost = lambda subset: 1 if constant_cost else len(subset)
            )
```

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

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

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

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

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

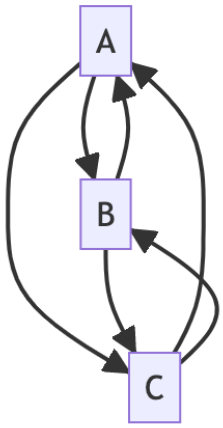


Figure 1: Fully connected graph

The heuristic function is slightly different:

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

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

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

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

```

```
    def flatten_list(self, _list):
        return list(itertools.chain.from_iterable(_list))

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    def get_number_of_elements_not_in_second_list(self, list1, list2):
        count = 0
        # flattened_list = self.flatten_list(list2)
        for i in set(list1):
            # print("i: ", i)
            if i not in list2:
                count += 1
        # if count > 1:
        #     print("count: ", count)
        return len(set(list1) - set(list2))

    # f(n) = h(n) + g(n)

    def h(self, n):
        num_new_elements =
        ↪  self.get_number_of_elements_not_in_second_list(self.list_values[n],
        ↪  self.discovered_elements)
        # if self.list_values[n] in self.final_list:
        #     return 1000
        return num_new_elements
        # return self.H[n] / (num_new_elements + 1)

    def get_node_with_least_h(self):
        min_h = float("inf")
        min_node = None
        for node in self.adjacency_list:
            if self.h(node) < min_h:
                min_h = self.h(node)
                min_node = node
        return min_node

    def get_node_with_least_h_and_not_in_final_list(self):
        min_h = float("inf")
        min_node = None
        for node in self.adjacency_list:
            if self.h(node) < min_h and node not in self.final_list:
                min_h = self.h(node)
                min_node = node
        return min_node

    # visited_node = [1, 2, 3]
    # final_list = [[4, 5], [1]]
    def are_we_done(self):
        # 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
```

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

                reconst_path.append(start_node)

                reconst_path.reverse()

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

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

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

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


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

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

2.2 Results

2.3 Received Reviews

Diego Mangasco

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

Ramin

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

Arman

Hi Sid,
here is my review:
The algorithm you tried as an augmented greedy solution is finding good solutions for small Ns, e.g. 29 for N=20 which is close to the exact solution. (you forgot to put N=20 in the solutions as well, it's good to add it as you are using this as your baseline). The function which it uses for cost is actually a kind of heuristic used in a greedy context. It is an interesting use case. for large Ns, It does not improve the solution, although meaningfully reduces the number of visited nodes. It's a kind of behaviour we observe when using heuristics in other search algorithms as well.
for A* search, your code is pretty clean and organised specially implementing in a class which makes it reusable. the heuristic is reasonable and simple. comparing length as cost and unit cost is useful to see the difference. My experience was that not using cost and not keeping parents did not made much difference in this specific problem and it makes code much smaller and faster.
The fact that you used the itertools methods has made your code cleaner and more elegant. It is better to implement loops, e.g. in are_we_done() using comprehension, using inner loops in separate line will affect the speed significantly.
Using a fully connected graph is interesting experiment, I will follow.
Bests

2.4 Given Reviews

2.4.1 Shayan

Shayan's code

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

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


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

Hi Shayan,

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

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

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

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

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

2.4.2 Arman

Arman's code

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

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

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


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

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

```
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
            ↪ covers N"""
53
54          current_numbers = {x for n in current_state.lists_ for x in ref_lists[n]}
55          return current_numbers == NUMBERS
56
57      def valid_actions(current_state:State,ref_lists):
58          """returns set of indexes not currently added to this state"""
59          return {indx for indx,_ in enumerate(ref_lists) if indx not in
            ↪ current_state.lists_}
60
61      def result(current_state,action):
62          next_state=State(current_state.copy_data()).add(action)
63          return next_state
64
65      def search(initial_state:State, ref_lists,priority_function:Callable):
66          frontier = PriorityQueue()
67          state = initial_state
68          state_count = 0
69          while state is not None and not goal_test(state,ref_lists):
70              for a in valid_actions(state,ref_lists):
71                  new_state = result(state,a)
72                  if new_state not in frontier:
73                      frontier.push(new_state,p=priority_function(new_state))
74                  elif new_state in frontier:
75                      pass
76              if frontier:
77                  state = frontier.pop()
78                  state_count+=1
79              else:
80                  state = None
81
82          logging.info(f"Found a solution with cost: {state.get_weight(ref_lists)}
            ↪ and {state_count} number of visited states, last state: {state}")
83
84      def heuristic(state:State,ref_lists,N):
85          remained = NUMBERS - state.get_items(ref_lists)
86          return len(remained) + random.randint(0,len(remained)//2)
87
```

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

Hi Arman,

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

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

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

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

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

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

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

3 Lab 2

3.1 Solution

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

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

1. Mutation: randomly change a 0 to a 1 or vice versa
2. Crossover: randomly select a point in the list and swap the values after that point

3.1.1 Representing the problem

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

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

3.1.2 Assessing Fitness

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

1. Potential duplicates
2. Undiscovered elements
3. Length of subset

The following equations were formulated for fitness assessment:

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

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

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

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

Table 1: Results of the algorithm

$$len(distinct\_elements)/(num\_undiscovered\_elements + 1) \tag{4}$$

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

3.2 Results

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

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

3.2.1 The Case of Mutations

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

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

```
    else:
        return 0.1

def plateau_detection(num_generations, fitness_log):
    '''
    Checks if the fitness has plateaued for the last num_generations.
    '''
    # this function is not used
    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

```
    def flip_mutation(genome, mutate_only_one_element=False):
        '''
        Flips random bit(s) in the genome.
        Parameters:
        mutate_only_one_element: If True, only one bit is flipped.
        '''
        modified_genome = genome.copy()
        if mutate_only_one_element:
            # flip a random bit
            index = random.randint(0, len(modified_genome) - 1)
            modified_genome[index] = 1 - modified_genome[index]
        else:
            # flip a random number of bits
            num_to_flip = choose_mutation_rate(fitness_log) * len(modified_genome)
            to_flip = random.sample(range(len(modified_genome)), int(num_to_flip))
            # to_flip = random.sample(range(len(modified_genome)), random.randint(0,
            ↪  len(modified_genome)))
            modified_genome = [1 - modified_genome[i] if i in to_flip else
            ↪  modified_genome[i] for i in range(len(modified_genome))]

        # mutate only if it brings some benefit to the weight
        # if calculate_weight(modified_genome) < calculate_weight(genome):
        #     return modified_genome

        return return_best_genome(modified_genome, genome)
```

3.3.2 Scramble Mutation

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

3.3.3 Swap Mutation

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

3.3.4 Inversion Mutation

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

3.4 Full Code

```python
import numpy as np
import itertools

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

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

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

len(initial_population)

```

```
fitness_log = []

def calculate_weight(genome):
    '''
    Weight Function
    Weight is the sum of the lengths of the subsets that are taken
    '''
    # select the subsets from prob based on the best individual
    final = [prob[i] for i, gene in enumerate(genome) if gene == 1]
    weight = len(list(itertools.chain.from_iterable(final)))
    return weight

def choose_mutation_rate(fitness_log):
    # choose mutation rate based on change in fitness_log
    if len(fitness_log) == 0:
        return 0.2
    if len(fitness_log) < 3:
        considered_elements = len(fitness_log)
    else:
        considered_elements = 3
    growth_rate = np.mean(np.diff(fitness_log[-considered_elements:]))
    if growth_rate <= 0:
        return 0.4
    elif growth_rate < 0.5:
        return 0.3
    elif growth_rate < 1:
        return 0.01
    else:
        return 0.1

def plateau_detection(num_generations, fitness_log):
    '''
    Checks if the fitness has plateaued for the last num_generations.
    '''
    if len(fitness_log) < num_generations:
        return False
    return all(fitness_log[-num_generations] == fitness_log[-i] for i in range(1,
    ↪  num_generations))

def flip_mutation(genome, mutate_only_one_element=False):
    '''
    Flips random bit(s) in the genome.
    Parameters:
    mutate_only_one_element: If True, only one bit is flipped.
    '''
    modified_genome = genome.copy()
    if mutate_only_one_element:
        # flip a random bit
        index = random.randint(0, len(modified_genome) - 1)
        modified_genome[index] = 1 - modified_genome[index]
```

```
    else:
        # flip a random number of bits
        num_to_flip = choose_mutation_rate(fitness_log) * len(modified_genome)
        to_flip = random.sample(range(len(modified_genome)), int(num_to_flip))
        # to_flip = random.sample(range(len(modified_genome)), random.randint(0,
        ↪  len(modified_genome)))
        modified_genome = [1 - modified_genome[i] if i in to_flip else
        ↪  modified_genome[i] for i in range(len(modified_genome))]

    return modified_genome
    # mutate only if it brings some benefit to the weight
    # if calculate_weight(modified_genome) < calculate_weight(genome):
    #     return modified_genome


def return_best_genome(genome1, genome2):
    return genome1
    # if calculate_fitness(genome1) > calculate_fitness(genome2):
    #     return genome1
    # else:
    #     return genome2

def mutation(genome):
    '''
    Runs a randomly chosen mutation on the genome. Mutations are:
    1. Bit Flip Mutation
    2. Scramble Mutation
    3. Swap Mutation
    4. Inversion Mutation
    Refer to README for more details.
    '''
    # check type of genome (debugging)
    # if type(genome) == tuple:
    #     print("genome is tuple")
    #     print(genome)

    possible_mutations = [flip_mutation, scramble_mutation, swap_mutation,
    ↪  inversion_mutation]
    chosen_mutation = random.choice(possible_mutations)
    return chosen_mutation(genome)

    # if random.random() < 0.1:
    #     for _ in range(num_elements_to_mutate):
    #         index = random.randint(0, len(genome) - 1)
    #         genome[index] = 1 - genome[index]
    # mutate a random number of elements
    # to_flip = random.randint(0, len(genome))
    # # flip the bits
    # return [1 - genome[i] if i < to_flip else genome[i] for i in
    ↪  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],
        ↪  len(modified_genome[start:end]))
153     return return_best_genome(modified_genome, genome)
154
155 def swap_mutation(genome):
156     '''
157     Randomly swaps two elements in the genome.
158     '''
159     modified_genome = genome.copy()
160     index1 = random.randint(0, len(modified_genome) - 1)
161     index2 = random.randint(0, len(modified_genome) - 1)
162     modified_genome[index1], modified_genome[index2] = modified_genome[index2],
        ↪  modified_genome[index1]
163     return return_best_genome(modified_genome, genome)
164
165 def inversion_mutation(genome):
166     '''
167     Randomly inverts the genome.
168     '''
169     modified_genome = genome.copy()
170     # select start and end indices to invert
171     start = random.randint(0, len(modified_genome) - 1)
172     end = random.randint(start, len(modified_genome) - 1)
173     # invert the elements
174     modified_genome = modified_genome[:start] + modified_genome[start:end][::-1]
        ↪  + modified_genome[end:]
175     return return_best_genome(modified_genome, genome)
176
177 def crossover(genome1, genome2):
178     '''
179     Crossover the two genomes by randomly selecting a point
180     '''
181     # crossover at a random point
182     crossover_point = random.randint(0, len(genome1))
183     modified_genome = genome1[:crossover_point] + genome2[crossover_point:]
184     return modified_genome
185
186 def roulette_wheel_selection(population):
187     '''
188     Selects an individual from the population based on the fitness.
```

```
    '''
    # calculate the total fitness of the population
    total_fitness = sum([individual.fitness[0] for individual in population])
    # select a random number between 0 and the total fitness
    random_number = random.uniform(0, total_fitness)
    # select the individual based on the random number
    current_fitness = 0
    for individual in population:
        current_fitness += individual.fitness[0]
        if current_fitness > random_number:
            return individual

def stochastic_universal_sampling(population):
    '''
    Select using Stochastic Universal Sampling.
    '''
    point_1 = random.uniform(0, 1)
    point_2 = point_1 + 1
    # In Progress

def rank_selection(population):
    '''
    Select using Rank Selection. Read more here:
    https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.h
    '''
    # sort the population based on the fitness
    population.sort(key=lambda x: x.fitness[0], reverse=True)
    # calculate the total rank
    total_rank = sum([i for i in range(len(population))])
    # select a random number between 0 and the total rank
    random_number = random.uniform(0, total_rank)
    # select the individual based on the random number
    current_rank = 0
    for i, individual in enumerate(population):
        current_rank += i
        if current_rank > random_number:
            return individual


def tournament(population, selection_method='tournament'):
    '''
    Selects the best individual from a random sample of the population.
    '''
    if selection_method == 'roulette':
        participant = roulette_wheel_selection(population)
        participant = Individual(participant.genome, participant.fitness)
    elif selection_method == 'rank':
        participant = rank_selection(population)
        participant = Individual(participant.genome, participant.fitness)
```

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

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

3.5 Received Reviews

s295103

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

s295103

Design considerations - Overall good solution, nice work trying multiple parent selection functions, different fitness functions, and using multiple mutation functions
Implementation considerations - After calling the problem() function it is necessary to reset the seed to a random value using ‘random.seed()‘ otherways all runs will always use 42 as seed value, so they won’t be truly random

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

```

should substituted by

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

```

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

```
    def *_mutation(genome):
        modified_genome = genome.copy()
        ...
        return modified_genome
```

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

can become

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

```
for i in range(NUM_GENERATIONS):
    # create offspring
    offspring = [generate(initial_population, i) for i in
    ↪  range(OFFSPRING_SIZE)]
```

Highlights/overall: The solution includes many different mutations which show an extra effort to improve the results with a broad approach. The change in the mutation rate based on the $fitness\_log$ is an interesting idea and seems to be effective. The code and results are very good!

3.6 Given Reviews

3.6.1 Erik

Erik's code

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


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

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

```
    return sorted_population[:POPULATION_SIZE]


def main():
    logging.basicConfig(level=logging.DEBUG)
    problem_space = problem(N, seed=42)
    population = select_rand_solution(problem_space)

    # should hold current population with the calculated fitness
    current_individuals = []

    # setup data structure, list of tuples containing ([entries], fitness) and sort
    for individual in population:
        current_individuals.append((individual, calculate_fitness(individual)))

    current_individuals = sorted(current_individuals, key=lambda l: l[1], reverse=True)

    counter = 0
    while counter < NR_OF_GENERATIONS:
        # a) Select individuals with a good fitness score for reproduction.
        cross_over_list = []
        for i in range(OFFSPRING_SIZE):
            parent_one = select_parents(current_individuals)
            parent_two = select_parents(current_individuals)

            # b) Let them produce offspring. Mutate with 5% chance
            tmp_child = crossover(parent_one, parent_two)
            if random.random() > 0.95:
                tmp_child = mutate_child(tmp_child, population)

            cross_over_list.append((tmp_child, calculate_fitness(tmp_child)))

        current_individuals = update_population(current_individuals, cross_over_list)
        counter += 1

    for solution in current_individuals:
        if goal_check(solution[0]):
            logging.info(f'Best solution for N={N} was {current_individuals[0][0]} \nWith a weight of {sum(len(_) for _ in current_individuals[0][0])}')
            break
```

Hi Eric,

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

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

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

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

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

Let's break down the solution itself:

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

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

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

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

```
  return miss-duplicates
```

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

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

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

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

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

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

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

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

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

3.6.2 Karl

Karl's code

```
  # helping functions

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

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

```
    :return: the count
    """
    list_elems = [single_elem for l in genome for single_elem in l]
    duplicates = sum([len(list(group))-1 for key, group in
    ↪  groupby(sorted(list_elems))])
    return duplicates
# to initialize the population
def create_population(STATE_SPACE, GOAL):
    """
    Initialize the population.
    :param STATE_SPACE: List of lists generated from problem-function
    :param GOAL: set of integers from 0 to N-1
    :return: a list of tuples: (genome,fitness), for each individual in the
↪   population.
    """
    population = []
    for _ in range(POPULATION_SIZE):
        individual = []
        for _ in range(random.randint(1,len(STATE_SPACE))):
            l = random.choice(STATE_SPACE)
            if l not in individual: #check duplicates here
                individual.append(l)
        #individual =
        ↪  random.choices(STATE_SPACE,k=random.randint(1,len(STATE_SPACE)))
        fitness = compute_fitness(individual, GOAL)
        population.append((individual,fitness))
    return population

def compute_fitness(genome, GOAL):
    """
    fitness is a tuple of (-#of_elems_missing,-#duplicates) which should be
↪   maximized
    :param genome: the sub-lists with random integers between 0 and N-1
    :param GOAL: set of integers from 0 to N-1
    :return: the fitness
    """
    # violated constraints, i.e. how many elements are missing
    vc = GOAL.difference(lists_to_set(genome))
    duplicates = count_duplicates(genome)
    # it is worse to lack elements than having duplicates
    fitness = (-len(vc), -duplicates)
    return fitness

def goal_check(genome, GOAL):
    """
    Check if all required elements are in the genome
    :param genome: the sub-lists with random integers between 0 and N-1
    :param GOAL: set of integers from 0 to N-1
    :return: boolean value if goal reached or not
    """
```

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

```
    :return: the mutated child
    """
    idx = random.randint(0,len(child))
    #child = child[:idx] + child[idx+1:] +
    ↪ STATE_SPACE[random.randint(0,len(STATE_SPACE)-1)]
    i = 0
    while i<10:
        i+=1
        if STATE_SPACE[random.randint(0,len(STATE_SPACE)-1)] not in child:
            child = child[:idx] + child[idx+1:] +
            ↪ STATE_SPACE[random.randint(0,len(STATE_SPACE)-1)]
            break
    return child

def update_population_plus(population, offspring):
    """
    Using the plus strategy to update population to next generation.
    :param population:
    :param offspring:
    :return: the best individuals in union(population, offspring)
    """
    tot = population + offspring
    ranked_population = sorted(tot, key=lambda t : t[1], reverse=True)
    return ranked_population[:POPULATION_SIZE]

def update_population_comma(offspring):
    """
    Using the plus strategy to update population to next generation.
    :param offspring:
    :return: the best individuals in from offspring
    """
    ranked_pop = sorted(offspring, key=lambda t : t[1], reverse=True)
    return ranked_pop[:POPULATION_SIZE]

def update_mutation_prob(best_solution, best_this_iter, mutation_param, it):
    """
    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).
    :param best_solution: The best solution so far
    :param best_this_iter: The best solution of this generation
    :param mutation_param:
    :param it: iteration number
    :return: the new mutation probability
    """
    if best_solution[1] >= best_this_iter[1]:
        mutation_param +=1
    elif best_solution[1] >= best_this_iter[1] and mutation_param>0:
        mutation_param -= 1
    return mutation_param/(1+it), mutation_param
```

```python
def solve_problem(N):
    STATE_SPACE = problem(N,seed=42)
    GOAL = set(range(N))
    population = create_population(STATE_SPACE, GOAL)
    best_sol = population[0] #to be updated after each iter
    found_in_iter = 0 #to be updated
    mutation_param = 1 #increase if solution doesn't improve
    mutation_prob = 0.1 #init value
    for i in range(ITERS):
        offspring = []
        for __ in range(OFFSPRING_SIZE):
            parent1, parent2 = parent_selection(population),
            ↪  parent_selection(population)
            child = cross_over(parent1,parent2, STATE_SPACE, mutation_prob)
            child_fitness = compute_fitness(child, GOAL)
            offspring.append((child,child_fitness))
        population = update_population_plus(population, offspring)
        #population = update_population_comma(offspring)
        best_curr = sorted(population, key=lambda l:l[1], reverse=True)[0]
        mutation_prob, mutation_param = update_mutation_prob(best_sol, best_curr,
        ↪  mutation_param, i)
        if goal_check(best_curr[0],GOAL) and best_curr[1] > best_sol[1]:
            best_sol = best_curr
            found_in_iter = i
    logging.info(f'Best solution found in {found_in_iter} iters and has weight
    ↪  {-best_sol[1][1]}')
    return best_sol
# main

# settings
POPULATION_SIZE = 50
OFFSPRING_SIZE = 30
ITERS = 100

for N in [5,10,20,50,100,1000,2000]:
    best_sol = solve_problem(N)
    print(f'N = {N}')
    logging.info(f'The best weight for N = {N}: {-best_sol[1][1]+N}')
```

Hi Karl,

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

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

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

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

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

Potential Improvements:

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

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

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

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

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

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

All in all, good job!

3.6.3 Ricardo

Ricardo's code

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

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

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

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

        w += list_length

    if cg_new_length == 0:
        goal_reached = True

    fitness = new_elements - repeated_elements

    return fitness, goal_reached, w


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

```

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

```
91      for g in range(GENERATIONS):
92          population = sorted(population, key=lambda i: i.fitness,
            ↪  reverse=True)[:POPULATION_SIZE-OFFSPRING_SIZE]
93
94          for i in range(OFFSPRING_SIZE):
95              p1 = select_parent(population,
                ↪  tournament_size=int(0.2*genome_size))
96              p2 = select_parent(population,
                ↪  tournament_size=int(0.2*genome_size))
97              o = cross_over(p1, p2, genome_size, list_of_lists, goal)
98              fitness, goal_reached, w = fitness_goal_eval(list_of_lists, o,
                ↪  goal)
99              o = mutation(o, genome_size, k=2)
100
101             population.append(Individual(o, fitness, goal_reached, w))
102
103
104
105     for i in population:
106         if i.goal_reached:
107             return i, population
108
109     print(f"No solution for current population (N={N})")
110     return None, population
111 N = 500
112 POPULATION_SIZE = 100
113 OFFSPRING_SIZE = 20
114 GENERATIONS = 200
115 PROB = 0.5
116
117 logging.getLogger().setLevel(logging.INFO)
118
119 solution, population = genetic_algorithm()
120 if solution != None:
121     logging.info(
122         f" Genetic algorithm solution for N={N:,}: "
123         + f"fitness={solution.fitness:,} "
124         + f"w={solution.w:,} "
125         + f"(bloat={solution.w/N*100:.0f}%)"
126     )
127 INFO:root: Genetic algorithm solution for N=500: fitness=-1,980 w=2,980
    ↪  (bloat=596%)
128 POPULATION_SIZE = 50
129 OFFSPRING_SIZE = 20
130 GENERATIONS = 200
131 PROB = 0.5
132
133 logging.getLogger().setLevel(logging.INFO)
134
135 for N in [5, 10, 20, 100, 500, 1000]:
```

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

Hi Ricardo,

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

Positives (both cosmetic and logical):

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

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

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

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

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

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

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

Things to look at:

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

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

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

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

All in all, good job!

3.6.4 Francesco

Francesco's code

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

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

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

  def intersection(lst1, lst2):
```

```
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)
```

```
                    if r not in g:
                        g.append(r)
                else:
                    g = []
            population.append(g)
        return [Individual(g, tuple((covering(g),-w(g)))) for g in population]
    N = 1000
    all_lists = problem(N,seed=42)
    Individual = namedtuple("Individual", ["genome", "fitness"])
    mu = 2000
    GENERATIONS = 100
    OFFSPRINGS_SIZE = 1100
    population = create_population(mu)

    for g in range(GENERATIONS):
        new_population = []
        for _ in range(OFFSPRINGS_SIZE):
            o = []
            if random.random() < 0.001:
                p = tournament(population)
                o = mutation(p.genome)
            else:
                p1 = tournament(population)
                p2 = tournament(population)
                o = cross_over(p1.genome, p2.genome)
            new_population.append(Individual(o, tuple((covering(o),-w(o)))))
        population += new_population
        population = sorted(population, key= lambda i : i.fitness,
        ↪  reverse=True)[:mu]

    print(f'w={w(population[0].genome)}, cov={covering(population[0].genome)}')
```

Hi Francesco,

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

Positives (both cosmetic and logical):

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

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

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

Some things to look at:

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

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

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

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

All in all, good job.

4 Lab 3

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

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

1. An agent using fixed rules based on nim-sum
2. An agent using evolved rules
3. An agent using minmax
4. An agent using reinforcement learning (both temporal difference learning and monte carlo learning)

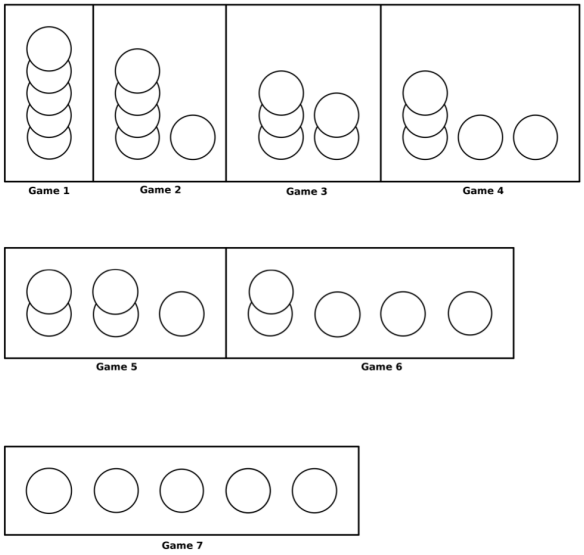
4.1 Solution

4.1.1 Fixed Rules

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

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

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



Player 1 has a winning strategy for all of these games! In game 1, the first player can just take all of the stones immediately. In games 2, 3, 4, and 5, the first player should use his first move to leave his opponent with two piles of the same size, and then mirror the opponents moves for the rest of the game (this will be explained in more detail in exercise 4). In games 6 and 7, the first player should use his first move to leave his opponent with four piles with one stone each; since they each can only take one stone for each of the next four turns, player 1 will win. □

Figure 2: Fixed Rules

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

from lib import Genome, Nim, Nimply


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

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

```
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
    ↪  from the smallest pile
39              4. If n piles and n-1 piles have the same size, remove x sticks from
    ↪  the smallest pile until it is the same size as the other piles
40              '''
41              population = []
42              for i in range(population_size):
43                  # rules 3 and 4 are fixed (apply for 3 or more piles)
44                  # different strategies for different rules (situations on the
                    ↪  board)
45                  individual = {
46                      'rule_1': [0, random.randint(0, (nim.num_rows - 1) * 2)],
47                      'rule_2a': [random.randint(0, 1), random.randint(0,
                        ↪  (nim.num_rows - 1) * 2)],
48                      'rule_2b': [random.randint(0, 1), random.randint(0,
                        ↪  (nim.num_rows - 1) * 2)],
49                      'rule_3': [nim.rows.index(min(nim.rows)), min(nim.rows)],
50                      'rule_4': [nim.rows.index(max(nim.rows)), max(nim.rows) -
                        ↪  min(nim.rows)]
51                  }
52                  genome = Genome(individual)
53                  population.append(genome)
54              return population
55
56          def statistics(self, nim: Nim):
57              '''
58              Similar to Squillero's cooked function to get possible moves
59              and statistics on Nim board
60              '''
61              # logging.info('In statistics')
62              # logging.info(nim.rows)
63              stats = {
64                  '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],
65                  # 'possible_moves': [(row, num_objects) for row in
                    ↪  range(nim.num_rows) for num_objects in range(1,
                    ↪  nim.rows[row]+1)],
66                  'num_active_rows': sum(o > 0 for o in nim.rows),
67                  'shortest_row': min((x for x in enumerate(nim.rows) if x[1] > 0),
                    ↪  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,
                        ↪  stats['possible_moves'][0][1]))
92                  elif stats["num_active_rows"] % 2 == 0:
93                      # logging.info('m2')
94                      if max(nim.rows) == 1:
95                          return Nimply(stats['longest_row'], 1)
96                      else:
97                          pile = random.choice([i for i, x in enumerate(nim.rows)
                            ↪  if x > 1])
98                          return Nimply(pile, nim.rows[pile] - 1)
99                  elif stats['num_active_rows'] == 3:
100                     # logging.info('m3')
101                     unique_elements = set(nim.rows)
102                     # check if 2 rows have the same number of sticks
103                     two_rows_with_same_elements = False
104                     for element in unique_elements:
105                         if nim.rows.count(element) == 2:
106                             two_rows_with_same_elements = True
107                             break
108
109                     if len(nim.rows) == 3 and two_rows_with_same_elements:
110                         # remove 1 stick from the longest row
111                         logging.info(nim.rows)
112                         return Nimply(stats['longest_row'], max(max(nim.rows) -
                            ↪  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
                                ↪  the same size as the other piles
124                             return Nimply(stats['shortest_row'],
                                ↪  max(nim.rows[stats['shortest_row']] -
                                ↪  counter.most_common()[1][0], 1))
125                     return random.choice(stats['possible_moves'])
126                 else:
127                     # logging.info('m5')
128                     return random.choice(stats['possible_moves'])
129             return engine
130 
131         def random_agent(self, nim: Nim):
132             '''
133             Random agent that takes a random move
134             '''
135             stats = self.statistics(nim)
136             return random.choice(stats['possible_moves'])
137 
138         def battle(self, opponent, num_games=1000):
139             '''
140             Battle this agent against another agent
141             '''
142             wins = 0
143             for _ in range(num_games):
144                 nim = Nim()
145                 while not nim.goal():
146                     nim.nimming_remove(*self.play(nim))
147                     if sum(nim.rows) == 0:
148                         break
149                     nim.nimming_remove(*opponent.play(nim))
150                 if sum(nim.rows) == 0:
151                     wins += 1
152             return wins
153 
154     if __name__ == '__main__':
155         rounds = 20
156         evolved_agent_wins = 0
157         for i in range(rounds):
158             nim = Nim(5)
159             orig = nim.rows
```

```
            fixedrule = FixedRuleNim()
            engine = fixedrule.strategy()

            # play against random
            player = 0
            while not nim.goal():
                if player == 0:
                    move = engine(nim)
                    logging.info('move of player 1: ', move)
                    nim.nimming_remove(*move)
                    player = 1
                    logging.info("After Player 1 made move: ", nim.rows)
                else:
                    move = fixedrule.random_agent(nim)
                    logging.info('move of player 2: ', move)
                    nim.nimming_remove(*move)
                    player = 0
                    logging.info("After Player 2 made move: ", nim.rows)
            winner = 1 - player
            if winner == 0:
                evolved_agent_wins += 1
        logging.info(f'Fixed rule agent won {evolved_agent_wins} out of {rounds}
        ↪  games')
```

Approach 2: Nim-Sum Will always win

```
from copy import deepcopy
from itertools import accumulate
from operator import xor
import random
import logging
from lib import Nim

# 3.1: Agent Using Fixed Rules
class ExpertNimSumAgent:
    '''
    Play the game of Nim using a fixed rule
    (always leave nim-sum at the end of turn)
    '''
    def __init__(self):
        self.num_moves = 0

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

```
        return result
        # return sum([i^r for i, r in enumerate(nim._rows)])

    def play(self, nim: Nim):
        # remove objects from row to make nim-sum 0
        nim_sum = self.nim_sum(nim)
        all_possible_moves = [(r, o) for r, c in enumerate(nim.rows) for o in
        ↪  range(1, c+1)]
        move_found = False
        for move in all_possible_moves:
            replicated_nim = deepcopy(nim)
            replicated_nim.nimming_remove(*move)
            if self.nim_sum(replicated_nim) == 0:
                nim.nimming_remove(*move)
                move_found = True
                break
        # if a valid move not found, return random move
        if not move_found:
            move = random.choice(all_possible_moves)
            nim.nimming_remove(*move)

        # logging.info(f"Move {self.num_moves}: Removed {move[1]} objects from
        ↪  row {move[0]}")
        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.

```
    '''
In this file, I will try to implement Nim where there is an evolved set of
↪  rules/strategies.
For each scenario, I will have a set of rules that will be used to determine the
↪  best move.
They are obtained from discussion with friends and from the paper "The Game of
↪  Nim" by Ryan Julian
The rules currently are:
1. If one pile, take $x$ number of sticks from the pile.
2. If two piles:
    a. If 1 pile has 1 stick, take x sticks
    b. If 2 piles have multiple sticks, take x sticks from the larger pile
3. If three piles and two piles have the same size, remove all sticks from the
↪  smallest pile
4. If n piles and n-1 piles have the same size, remove x sticks from the smallest
↪  pile until it is the same size as the other piles
'''

from collections import Counter, namedtuple
from copy import deepcopy
from itertools import accumulate
import logging
from operator import xor
import random
from typing import Callable

from lib import Genome, Nim, Nimply

class BrilliantEvolvedAgent:
    def __init__(self):
        self.num_moves = 0
        self.OFFSPRING_SIZE = 200
        self.POPULATION_SIZE = 50
        self.GENERATIONS = 100
        self.nim_size = 5

    def nim_sum(self, nim: Nim):
        '''
        Returns the nim sum of the current game board
```

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

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

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

```
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
```

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

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

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

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

```
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
    ↪  agent
263         (cannot use nim sum agent as it is too good)
264         '''
265         wins = 0
266         for i in range(5):
267             nim = Nim(5)
268             player = 0
269             engine = self.strategy(genome)
270             while not nim.goal():
271                 if player == 0:
272                     move = engine(nim)
273                     nim.nimming_remove(*move)
274                     player = 1
275                 else:
276                     nim.nimming_remove(*self.random_agent(nim))
277                     player = 0
278             winner = 1 - player
279             if winner == 0:
280                 wins += 1
281         return wins / 5
282 
283     def select_survivors(self, population: list, num_survivors: int):
284         '''
285         Select the best genomes from the population
286         '''
287         return sorted(population, key=lambda x: x.fitness,
        ↪  reverse=True)[:num_survivors]
288 
289     def learn(self, population_size=100, mutation_rate=0.1, crossover_rate=0.7,
    ↪  nim: Nim = None):
290         initial_population = self.init_population(population_size, nim)
```

```
        for genome in initial_population:
            genome.fitness = self.calculate_fitness(genome)
        for i in range(self.GENERATIONS):
            # logging.info(f'Generation {i}')
            new_offspring = []
            for j in range(self.OFFSPRING_SIZE):
                parent1 = random.choice(initial_population)
                parent2 = random.choice(initial_population)
                child = self.crossover(parent1, parent2, crossover_rate)
                child = self.mutate(child)
                new_offspring.append(child)
            initial_population += new_offspring
            initial_population = self.select_survivors(initial_population,
            ↪  population_size)
        best_strategy = initial_population[0]
        return best_strategy

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

if __name__ == '__main__':
    rounds = 20
    evolved_agent_wins = 0
    for i in range(rounds):
        nim = Nim(5)
        orig = nim.rows
        brilliantagent = BrilliantEvolvedAgent()
        best_strategy = brilliantagent.learn(nim=nim)
        engine = brilliantagent.strategy(best_strategy)

        # play against random
        player = 0
        while not nim.goal():
            if player == 0:
                move = engine(nim)
                logging.info('move of player 1: ', move)
                nim.nimming_remove(*move)
```

```
                player = 1
                logging.info("After Player 1 made move: ", nim.rows)
            else:
                move = brilliantagent.random_agent(nim)
                logging.info('move of player 2: ', move)
                nim.nimming_remove(*move)
                player = 0
                logging.info("After Player 2 made move: ", nim.rows)
        winner = 1 - player
        if winner == 0:
            evolved_agent_wins += 1
    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.*

```
thresholds = [p1, p2, p3]
if random.random() < p1:
    # strategy 1...
elif random.random() < p2:
    # strategy 2...
else:
    # strategy 3...

class GA:
    ...

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.*

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

```

```
    from lib import Nim

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

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

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

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

```
50                  strategy =
                    ↪ random.choice([strategies['all_elements_shortest_row'],
                    ↪ strategies['1_element_shortest_row'],
                    ↪ strategies['random_element_shortest_row']])
51              elif p >= prob_list[1] and p < prob_list[2]:
52                  strategy = random.choice([strategies['all_elements_longest_row'],
                    ↪ strategies['1_element_longest_row'],
                    ↪ strategies['random_element_longest_row']])
53              else:
54                  strategy = strategies['nim_sum']
55
56              nim.nimming_remove(*strategy)
57              self.num_moves += 1
58              return sum(nim.rows)
59
60          def play(self, nim: Nim):
61              '''
62              Play the game of Nim using the evolved rules
63              '''
64              prob_list = [0.25, 0.5, 0.75, 1]
65              prob_list = self.evolve_probabilities(nim, prob_list, 20, 5)
66              self.play_nim(nim, prob_list)
67
68          def crossover(self, p1, p2):
69              '''
70              Crossover between two parents
71              '''
72              return np.random.choice(p1 + p2, size=4, replace=True)
73
74          def evolve_probabilities(self, nim: Nim, prob_list: list,
            ↪ num_generations: int, num_children: int):
75              '''
76              Evolve the probabilities of the strategies
77              '''
78              # create initial population
79              population = [prob_list for _ in range(num_children)]
80              # create initial fitness scores
81              fitness_scores = [self.play(nim, p) for p in population]
82              # create initial parents
83              parents = [population[i] for i in np.argsort(fitness_scores)[:2]]
84              # create new population
85              new_population = []
86              for _ in range(num_generations):
87                  # create children
88                  for _ in range(num_children):
89                      p1 = random.choice(parents)
90                      p2 = random.choice(parents)
91                      child = self.crossover(p1, p2)
92                      # child = []
93                      # for i in range(len(parents[0])):
```

```
                    #     # crossover between parents

                    #     child.append(random.choice(parents)[i])
                    new_population.append(child)
                # create fitness scores
                fitness_scores = [self.play_nim(nim, p) for p in new_population]
                # create new parents
                parents = [new_population[i] for i in
                ↪  np.argsort(fitness_scores)[:2]]
                # create new population
                new_population = []
            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.

```
from copy import deepcopy
from functools import lru_cache
from itertools import accumulate
import math
from operator import xor
from evolved_nim import BrilliantEvolvedAgent
import logging
from lib import Nim

logging.basicConfig(level=logging.INFO)

class MinMaxAgent:
    def __init__(self):
        self.num_moves = 0

    def nim_sum(self, nim: Nim):
        '''
        Returns the nim sum of the current game board
        by taking an XOR of all the rows.
```

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

    def evaluate(self, nim: Nim, is_maximizing: bool):
        '''
        Returns the evaluation of the current game board
        '''
        if all(row == 0 for row in nim.rows):
            return -1 if is_maximizing else 1
        else:
            return -1

    @lru_cache(maxsize=1000)
    def minmax(self, nim: Nim, depth: int, maximizing_player: bool, alpha: int =
    ↪  -1, beta: int = 1, max_depth: int = 7):
        '''
        Depth-limited Minimax algorithm to find the best move with alpha-beta
↪   pruning and depth limit
        '''
        logging.info("Depth ", depth)
        if depth == 0 or nim.goal() or depth == max_depth:
            # logging.info("Depth ", depth)
            # logging.info("Nim goal ", nim.goal())
            return self.evaluate(nim, maximizing_player)

        if maximizing_player:
            value = -math.inf
            for r, c in enumerate(nim.rows):
                for o in range(1, c+1):
                    # make copy of nim object before running a nimming operation
                    replicated_nim = deepcopy(nim)
                    replicated_nim.nimming_remove(r, o)
                    value = max(value, self.minmax(replicated_nim, depth-1,
                    ↪  False, alpha, beta))
                    alpha = max(alpha, value)
                    if beta <= alpha:
                        logging.info("Pruned")
                        break
            return value
        else:
            value = math.inf
            for r, c in enumerate(nim.rows):
                for o in range(1, c+1):
                    # make copy of nim object before running a nimming operation
                    replicated_nim = deepcopy(nim)
                    replicated_nim.nimming_remove(r, o)
                    value = min(value, self.minmax(replicated_nim, depth-1, True,
                    ↪  alpha, beta))
```

```
                    beta = min(beta, value)
                    if beta <= alpha:
                        logging.info("Pruned")
                        break
            return value

    def play(self, nim: Nim):
        '''
        Agent returns the best move based on minimax algorithm
        '''
        possible_moves = []
        for r, c in enumerate(nim.rows):
            for o in range(1, c+1):
                # make copy of nim object before running a nimming operation
                replicated_nim = deepcopy(nim)
                replicated_nim.nimming_remove(r, o)
                possible_moves.append((r, o, self.minmax(replicated_nim, 10,
                ↪  False)))
        # sort possible moves by the value returned by minimax
        possible_moves.sort(key=lambda x: x[2], reverse=True)
        # return the best move
        return possible_moves[0][0], possible_moves[0][1]

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

if __name__ == "__main__":

    rounds = 10

    minmax_wins = 0
    for i in range(rounds):
        nim = Nim(num_rows=5)
        agent = MinMaxAgent()
        random_agent = BrilliantEvolvedAgent()
        player = 0
        while not nim.goal():
```

```
            if player == 0:
                move = agent.play(nim)
                logging.info(f"Minmax move {agent.num_moves}: Removed {move[1]}
                ↪  objects from row {move[0]}")
                logging.info(nim.rows)
                nim.nimming_remove(*move)
            else:
                move = random_agent.random_agent(nim)
                logging.info(f"Random move {random_agent.num_moves}: Removed
                ↪  {move[1]} objects from row {move[0]}")
                logging.info(nim.rows)
                nim.nimming_remove(*move)
            player = 1 - player

        winner = 1 - player
        if winner == 0:
            minmax_wins += 1
        # player that made the last move wins
        logging.info(f"Player {winner} wins in round {i+1}!")

    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.

```
class NimRLTemporalDifferenceAgent:
"""
An agent that learns to play Nim through temporal difference learning.
"""
def __init__(self, num_rows: int, epsilon: float = 0.4, alpha: float = 0.3,
    gamma: float = 0.9):
    """Initialize agent."""
    self.num_rows = num_rows
    self.epsilon = epsilon
    self.alpha = alpha
    self.gamma = gamma
    self.current_state = None
    self.previous_state = None
    self.previous_action = None
    self.Q = dict()

def init_reward(self, state: Nim):
    '''Initialize reward for every state and every action with a random value'''
    for i in range(1, state.num_rows):
        nim = Nim(num_rows=i)
        for r, c in enumerate(nim.rows):
            for o in range(1, c+1):
                self.set_Q(hash_list(nim.rows), (r, o),
                           np.random.uniform(0, 0.01))

def get_Q(self, state: Nim, action: tuple):
    """Return Q-value for state and action."""
    if (hash_list(state.rows), action) in self.Q:
        logging.info("Getting Q for state: {} and action:
        {}".format(hash_list(state.rows), action))
        logging.info("Q-value: {}".format(self.Q[(hash_list(state.rows),
        action)]))
        return self.Q[(hash_list(state.rows), action)]
    else:
        # initialize Q-value for state and action
        self.set_Q(hash_list(state.rows), action, np.random.uniform(0, 0.01))
        return self.Q[(hash_list(state.rows), action)]

def set_Q(self, state: str, action: tuple, value: float):
    """Set Q-value for state and action."""
    # logging.info("Setting Q for state: {} and action: {} to value:
    {}".format(state, action, value))
    self.Q[(state, action)] = value

def get_max_Q(self, state: Nim):
    """Return maximum Q-value for state."""
    max_Q = -math.inf
    # logging.info(state.rows)
    for r, c in enumerate(state.rows):
        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
                    ↪  {}".format(state.rows, (r, o), val))
91                  self.set_Q(hash_list(state.rows), (r, o), val)
92              else:
93                  logging.info("State already registered: {} and action:
                    ↪  {}".format(state.rows, (r, o)))
94
```

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

```
        winner = 1 - player
        if winner == 0:
            wins += 1

    logging.info(f"Win Rate in round {round}: {wins / rounds}")

def battle(self, agent, rounds=1000, training=True, momentary_testing=False):
    """Train agent by playing against other agents."""
    agent_wins = 0
    winners = []
    for episode in range(rounds):
        # logging.info(f"Episode {episode}")
        nim = Nim(num_rows=5)
        self.current_state = nim
        self.previous_state = None
        self.previous_action = None
        player = 0
        while True:
            reward = 0
            if player == 0:
                self.previous_state = deepcopy(self.current_state)
                self.previous_action = self.get_action(self.current_state)
                self.current_state.nimming_remove(
                    *self.previous_action)
                player = 1
            else:
                move = agent(self.current_state)
                # logging.info("Random agent move: {}".format(move))
                self.current_state.nimming_remove(*move)
                player = 0

            # learning by calculating reward for the current state
            if self.current_state.goal():
                winner = 1 - player
                if winner == 0:
                    logging.info("Agent won")
                    agent_wins += 1
                    reward = 1
                else:
                    logging.info("Random won")
                    reward = -1
                winners.append(winner)
                self.update_Q(reward, self.current_state.goal())
                break
            else:
                self.update_Q(reward, self.current_state.goal())

        # decay epsilon after each episode
        self.epsilon = self.epsilon - 0.1 if self.epsilon > 0.1 else 0.1
        self.alpha *= -0.0005
```

```
        if self.alpha < 0.1:
            self.alpha = 0.1

        if training and momentary_testing:
            if episode % 100 == 0:
                logging.info(f"Episode {episode} finished, sampling")
                random_agent = BrilliantEvolvedAgent()
                self.test_against_random(
                    episode, random_agent.random_agent)

    if not training:
        logging.info("Reinforcement agent won {} out of {} games".format(
            agent_wins, rounds))
    # self.print_best_action_for_each_state()
    return winners

def choose_action(self, state: Nim):
    """Return action based on greedy policy."""
    max_Q = -math.inf
    best_action = None
    for r, c in enumerate(state.rows):
        for o in range(1, c+1):
            Q = self.get_Q(state, (r, o))
            if Q > max_Q:
                max_Q = Q
                best_action = (r, o)
    if best_action is None:
        return random.choice(self.get_possible_actions(state))
    else:
        return best_action

if __name__ == "__main__":
rounds = 10000
minmax_wins = 0

nim = Nim(num_rows=5)
agent_tda = NimRLTemporalDifferenceAgent(num_rows=5)
random_agent = RandomAgent()

# agentG = NimRLMonteCarloAgent(num_rows=7)
agent_tda.battle(random_agent.play, rounds=10000)
agent_tda.epsilon = 0.1

# TESTING
logging.info("Testing against random agent")
agent_tda.battle(random_agent.random_agent, training=False, rounds=1000)
```

Monte Carlo Learning

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

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

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

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


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


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


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

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

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

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

    def learn(self):
        target = 0

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

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

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

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

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

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

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

4.2 Acknowledgements

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!

Francesco Sattolo

Design considerations:

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

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

4.4 Given Reviews

4.4.1 Karl

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

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

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

    # ...

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

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

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

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

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

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

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

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

```
                    ply = Nimply(row, elem)

                # rule 5
                else:
                    row, elem = agent.rule5(data)
                    ply = Nimply(row, elem)


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

            return ply

        return evolvable


    # human strategy, make moves through input
    def my_strategy(state: Nim) -> Nimply:
        print(f'Current state: {state.rows}')
        data = cook_status(state)
        pm = data['possible_moves']
        index = input(f'Choose a play: {[(i, m) for i, m in enumerate(pm)]}')
        while True:
            try:
                assert int(index) in range(len(pm))
            except Exception:
                print('Invalid input, try again')
                index = input(f'Choose a play: {[(i, m) for i, m in
                ↪  enumerate(pm)]}')
            else:
                row = pm[int(index)][0]
                elems = pm[int(index)][1]
                break
        return Nimply(row, elems)


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


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

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

```
181         return best_contestor
182 
183 
184     def cross_over(parent1: Evolvable_agent, parent2: Evolvable_agent,
    ↪  mutation_prob: float) -> Evolvable_agent:
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 alwasy minimizes the opponents maximum win
255         Play against optimal strategy, should be able to win if start
256         Build as class or function?
257         Need:
258             keep value for each state (exhaustive)
259             condition: return 1 if win -1 else
260             condition: return 0 if not decided
261                 play intil determined and traverse back to that state
262     """
263     # %% MINMAX fcn
264     def minmax(state: Nim, my_turn: bool, alpha=-1, beta=1):
265         if not state:  # empty board then I lose
266             return -1 if my_turn else 1
267 
268         data = cook_status(state)
269         possible_new_states = []
270         for ply in data['possible_moves']:
271             tmp_state = deepcopy(state)
272             tmp_state.nimming(ply)
273             possible_new_states.append(tmp_state)
274         if my_turn:
```

```
            bestVal = -np.inf
            for new_state in possible_new_states:
                value = minmax(new_state, False, alpha, beta)
                bestVal = max(bestVal, value)
                alpha = max(alpha, bestVal)
                if beta <= alpha:
                    logging.info(f'Pruned')
                    break
            return bestVal
        else:
            bestVal = np.inf
            new_state = deepcopy(state)
            ply = optimal_strategy(new_state)
            new_state.nimming(ply)
            value = minmax(new_state, True, alpha, beta)
            bestVal = min(bestVal, value)
            return bestVal

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

    # %% Q4 - RL

    """
    Reinforcement learning agent to play Nim

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

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

    Outline:
        1. Selection (select an unvisited node) with highest UCT
        2. Expand to that node
        3. Simulate from that node until termination
        4. Backpropagate and update node with statistics
```

```
            * N(v) - number of visits for node v
            * Q(v) - value/reward playing from that node

    UCT:
        uct(v_i, v) = Q(v_i)/N(v_i) + c*sqrt(log(N(v))/N(v_i)), which prefers child nodes with small N(v_i)
        choose action according to highest uct value (init with np.inf to explore every move)
    """

    # Imports
    import itertools


    # Class

    class RLAgent:

        # INITIALIZATION
        # ------------------------------------------------------------------
        def __init__(self, nim_size: int, random_factor=0.2,
                     exploration_factor=np.sqrt(2)):  # explore with 20%, exploit with 80%
            self.nim_size = nim_size
            self.current_state = None
            self.previous_state = None
            self.__init_states(nim_size)
            self.random_factor = random_factor
            self.c = exploration_factor

        def __init_states(self, nim_size: int):
            """find all possible board positions"""
            states = {}
            rows = [i * 2 + 1 for i in range(nim_size)]
            elem_ranges = list(itertools.combinations([range(n + 1) for n in rows], r=nim_size))
            all_states = list(itertools.product(*elem_ranges[0]))

            for state in all_states:
                states[state] = {}
                states[state]['visits'] = 0
                states[state]['value'] = 0
                states[state]['child_states'] = self.__init_child_states(state)
            self.states = states
            # last state is the initial board
            self.current_state = all_states[-1]
            self.states[self.current_state]['visits'] = 1

        def __init_child_states(self, state):
            """Find all states accessible from state"""
```

```
            nim = Nim(self.nim_size)
            nim._rows = list(state)
            if nim:
                data = cook_status(nim)
                children = []
                for ply in data['possible_moves']:
                    tmp_nim = deepcopy(nim)
                    tmp_nim.nimming(ply)
                    children.append(tmp_nim.rows)
                return children

        # MCTS ------------------------------------------------------------------
        def selection(self):
            """Select next move according to highest uct score"""
            next_state = self.__state_with_highest_uct()
            return next_state

        def __state_with_highest_uct(self):
            """Move to child node with highest UCT score (depending on parent and
            ↪  child nodes) """
            visits_parent = self.states[self.current_state]['visits']
            best_state = None
            best_uct = -np.inf
            for child_state in self.states[self.current_state]['child_states']:
                visits_child = self.states[child_state]['visits']
                wins_child = self.states[child_state]['value']
                uct = wins_child / (visits_child + 1) + self.c *
                ↪  (np.log(visits_parent) / (visits_child + 1)) ** (1 / 2)
                if uct > best_uct:
                    best_uct = uct
                    best_state = child_state
            return best_state

        def random_selection(self):
            """Explore and move to random state"""
            next_state =
            ↪  random.choice(tuple(self.states[self.current_state]['child_states']))
            return next_state

        def expand(self, next_state):
            """Expand to the found next state. Return the ply that takes agent
            ↪  there"""
            self.previous_state = self.current_state
            self.current_state = next_state
            ply = self.__next_ply()
            return ply

        def __next_ply(self):
            """ Find ply that takes agent from previous state to current
            ↪  state"""
```

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

```
                        if random.random() < self.random_factor:
                            # choose random state
                            ns = self.random_selection()
                        else:
                            ns = self.selection()
                        ply = self.expand(next_state=ns)
                        nim.nimming(ply)

                        self.simulate(opponent, n_matches)

    def get_statistics(self):
        """Print overview of number of visits and wins for a visited
        ↪  state"""
        info = [(k, v['value'], v['visits']) for k, v in self.states.items()]
        for state in info:
            if state[2] > 0:  # at least 1 visit
                print(f'State {state[0]}: \tvisits {state[2]} \twins
                ↪  {state[1]}')

    def policy(self, state: Nim) -> Nimply:
        """The policy, i.e. the next move for the current state"""
        self.current_state = state.rows
        ns = self.selection()
        ply = self.expand(next_state=ns)
        return ply

# %% MAIN
import argparse

if __name__ == '__main__':

    # VARIABLES
    NIM_SIZE = 3
    NUM_MATCHES = 100
    EVAL_MATCHES = 100

    # INPUT
    parser = argparse.ArgumentParser()
    parser.add_argument("-t", "--task", dest="task", default=1,
                        help="Which task should run? Choose from 1, 2, 3 or
                        ↪  4.", type=int)

    args = parser.parse_args()
    print(f"Task: {args.task}")

    # ----------------------------TASK 1 - PLAYING THE OPTIMAL STRATEGY
    ↪  ----------------------------
    if args.task == 1:
        play_nim(optimal_strategy, optimal_strategy)
        # play the nim-sum strategy
```

```
503             starting_wins = evaluate(optimal_strategy, optimal_strategy)
504             print(f'Optimal strategy wins {starting_wins * 100: .0f}% when
                ↪  starting and {(1 - starting_wins) * 100: .0f}% when not
                ↪  starting.')
505
506         # ---------------------------TASK 2 - EVOLVE AN AGENT
            ↪  ---------------------------
507         elif args.task == 2:
508             # set params
509             POPULATION_SIZE = 50
510             OFFSPRING_SIZE = 200
511             GENERATIONS = 10
512             OPPONENTS = [dumb_agent, pure_random, semi_smart, optimal_strategy]
513
514             tournament_size = 10
515             mutation_prob = 0.3
516
517             pop = init_population()
518
519             for gen in tqdm(range(GENERATIONS), desc='Generations'):
520                 calc_fitness(pop)
521                 offspring = create_offspring(pop, tournament_size, mutation_prob)
522                 pop = get_next_generation(offspring)
523
524         # --------------------------- TASK 3 - MINMAX FUNCTION
            ↪  ---------------------------
525         elif args.task == 3:
526             import time
527             start = time.time()
528             play_nim(best_move, optimal_strategy)
529             elapsed = time.time() - start
530             print(f'It take {elapsed :.2f} seconds to play a game of Nim with
                ↪  size {NIM_SIZE}')
531
532         # --------------------------- TASK 4 - REINFORCEMENT LEARNING
            ↪  ---------------------------
533         elif args.task == 4:
534             ITERS = 1000
535
536             # must have run with -t 2 to have a pop
537             if 'pop' in locals():
538                 opponents = [pure_random, semi_smart, make_strategy(pop[0]),
                    ↪  optimal_strategy]
539             else:
540                 opponents = [pure_random, semi_smart, optimal_strategy]
541
542             for opponent in opponents:
543                 rl_agent = RLAgent(NIM_SIZE)
544                 rl_agent.learn_to_play([opponent], n_sims=ITERS,
                    ↪  n_matches=NUM_MATCHES)
```

```
                evaluate(rl_agent.policy, opponent)


        else:
            print(f'Have not finished task {args.task}')
```

Hi Karl,

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

1. There is a single file with the solutions for all labs. To improve readability, consider modularising by having a shared library file and a class for each task.

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

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

Evolutionary Algorithm

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

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

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

Minimax

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

Reinforcement Learning

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

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

Overall, good job!

Best, Sidharrth

5 Final Project

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

5.1 My Strategy For Solving the Problem

5.1.1 Step 1: Implement and Tune Multiple Search Algorithms

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

- **Random**: This agent randomly selects positions and pieces on the board. In the spirit of true randomness, it does not take into account the current state of the board.
- **Parameterized Hardcoded Play**: This agent has a set of fixed rules, where it attempts to build a line of like pieces. Where it cannot, it attempts
- **Deep Q-Learning**: This agent uses a deep neural network to approximate the Q-function. It uses a replay buffer to store the experience tuples and uses a target network to stabilize the training process. The agent uses an epsilon-greedy policy to balance exploration and exploitation. The input to the linear network is a flatten list of 1x16 pieces based on the current board composition, while the input to the convolutional neural network is a 4x4x4 board composition.
- **Q-Learning (Temporal Difference Learning)**: This agent uses a Q-table to store the Q-values. It uses a replay buffer to store the experience tuples and uses a target network to stabilize the training process. The agent uses an epsilon-greedy policy to balance exploration and exploitation. A custom OpenAI Gym environment is created to make training easier.
- **Monte Carlo Tree Search**: This agent uses a Monte Carlo Tree Search algorithm to select the best move. It uses a UCB1 formula to select the best child node at each iteration.
- **QL-MCTS**: This algorithm uses a Q-table as it's base and uses a rolled out Monte Carlo Search Tree for a more efficient search. When a state cannot

be found in the Q-table, the agent once again goes to the Monte Carlo Tree Search algorithm to find the best move.

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

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

2. **Deep Q-Learning (Linear and Convolutional Neural Network)**: In this approach, I train a 4-layer deep neural network to predict the Q-values of a given state. Despite several hours of training and hyperparameter tuning (changing the number of layers, optimiser, learning rate), the agent could only reach a 60% win rate in its best attempt. I also tried a convolutional neural network to feed the entire board composition as a 4x4x4 input (third dimension is the piece attribute), but training was far too slow.

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

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

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

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

5.1.2 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.

5.1.3 Step 3: Implementing the Hybrid Agent

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

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

$$couples + 2 * triplets$$

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

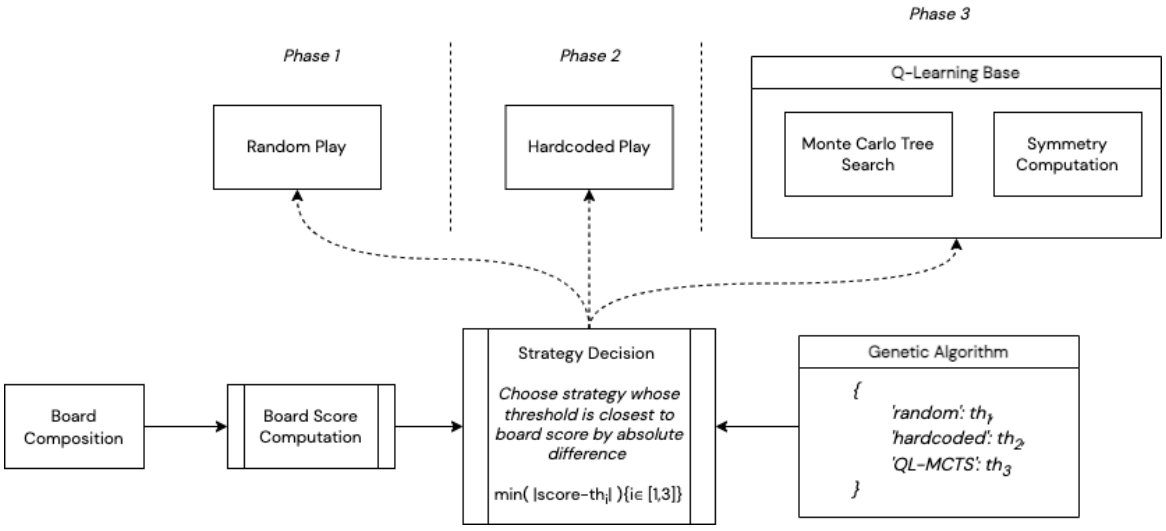


Figure 3: Hybrid Agent

Listing 1 Genome Example

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

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:

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

5.2 Code

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

5.3 Acknowledgements

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

- After I tried to get a working Deep Q-Network and realised that it wasn't

converging in reasonable time, we discussed the possibility of using some tree search algorithm. It was Diego who suggested MCTS.

- When realising that MCTS can be quite slow towards the end of the game, I suggested building a hybrid QL-MCTS player that would use a base Q-table to remember the best moves so the quadratically complex tree wouldn't need to store so many nodes.
- We later built a hardcoded agent using different rules and realised that it performed very well, and was quick to make a move.
- We then decided to combine everything we did into a hybrid agent that would switch between strategies depending on the board score.
- Diego suggested a good scoring function. He also suggested that a genetic algorithm could be used for this.
- I suggested finding score thresholds to switch between strategies using a genetic algorithm.

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

6 Conclusion

Ok bye.