Politecnico di Torino

01URRSM

Computational Intelligence Final Report

Sidharrth Nagappan

s307031

Contents

1 Introduction

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

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

2 Lab 1

2.1 Solution

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

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

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

2.1.3 A* Search Using a Priority Queue

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

The implemented algorithm can be surmised as pseudocode below:

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

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

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

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

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

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

    def h(self, state):
```

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

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

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

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

```python
                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 = $[1, 2]$,

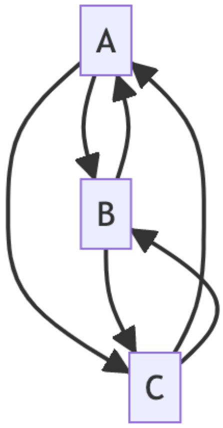

Figure 1: Fully connected graph

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()

```

```python
    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))
```

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

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

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

        open_list = [start_node]
        closed_list = []

        g = {}

        g[start_node] = 0

        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

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

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

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

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

```

```
                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

3 Lab 2

3.1 Solution

3.2 Results

3.3 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.

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

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

3.4 Given Reviews

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.

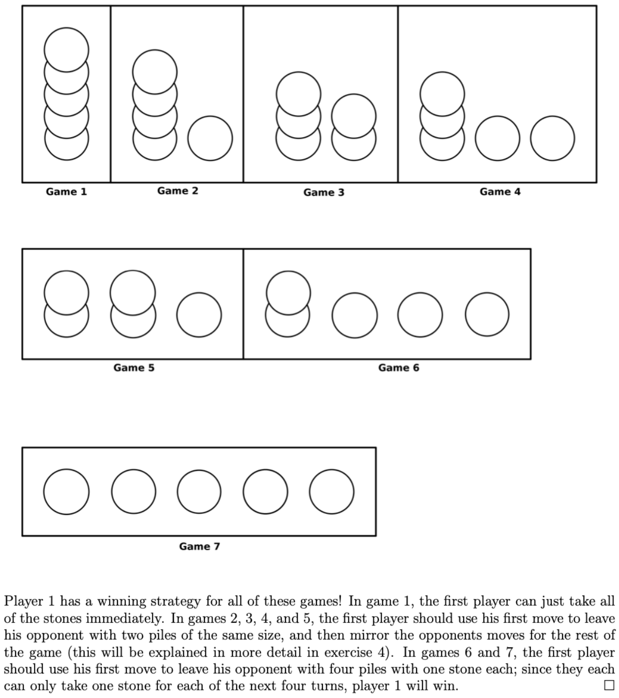

Figure 2: Fixed Rules

```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
```

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

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

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

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

        return stats

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

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

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

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

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

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

```python
            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

```python
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)
```

```python
        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.

```python
    '''
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]
```

```python
        return Genome(child)

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

    def mutate(self, genome: Genome, mutation_rate=0.5):
        '''
        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)
        '''
        rule = random.choice(list(genome.rules.keys()))
        # swap some keys
        if rule == 'rule_1':
            genome.rules[rule] = [0, random.randint(0, (self.nim_size - 1) * 2)]
        elif rule == 'rule_2a':
            genome.rules[rule] = [random.randint(0, 1), random.randint(0, (self.nim_size - 1) * 2)]
        elif rule == 'rule_2b':
            genome.rules[rule] = [random.randint(0, 1), random.randint(0, (self.nim_size - 1) * 2)]
        elif rule == 'rule_3':
            genome.rules[rule] = [random.randint(0, self.nim_size - 1), random.randint(0, (self.nim_size - 1) * 2)]
        elif rule == 'rule_4':
            genome.rules[rule] = [random.randint(0, self.nim_size - 1), random.randint(0, (self.nim_size - 1) * 2)]
        return genome
        # rule = random.choice(list(genome.rules.keys()))
        # if random.random() < mutation_rate:
        #     genome.rules[rule] = [random.randint(0, 1), random.randint(0, self.nim_size * 2)]
        # return genome
        # rule = random.choice(list(genome.keys()))
        # genome[rule] = random.randint(1, 10)

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

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

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

        return stats

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

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

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

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

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

```
                else:
                    # do something random
                    return Nimply(*random.choice(stats['possible_moves']))

            counter = Counter()
            for element in nim.rows:
                counter[element] += 1
            if len(counter) == 2:
                if counter.most_common()[0][1] == 1:
                    # remove x sticks from the smallest pile until it is the same size as the other piles
                    return Nimply(stats['shortest_row'], max(nim.rows[stats['shortest_row']] - counter.most_common()[1][0], 1))
                # else:
                #     return random.choice(stats['possible_moves'])

            # for large number of piles, general rule to remove all but 1 stick from a random pile
            if stats["num_active_rows"] % 2 == 0:
                if nim.rows[stats['longest_row']] == 1:
                    return Nimply(stats['longest_row'], 1)
                else:
                    pile = random.choice([i for i, x in enumerate(nim.rows) if x > 1])
                    return Nimply(pile, nim.rows[pile] - 1)

            else:
                # this is a fixed rule, does not have random component
                # rule from the paper Ryan Julian: The Game of Nim
                # 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
                # check if only 1 pile has a different number of sticks
                # just make a random move if all else fails
                return random.choice(stats['possible_moves'])
        return evolution

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

    def dumb_agent(self, nim: Nim):
        '''
        Agent that takes one element from the longest row
        '''
        stats = self.statistics(nim)
```

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

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

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

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

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

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

```
        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)
```

```python
            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.*

```python
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.*

```python
# 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]:
```

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

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

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

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

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

```
                    #     # 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():
```

```python
            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.

```python
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):
```

```
            # logging.info("Just Q: {}".format(self.get_Q(state, (r, o))))
            max_Q = max(max_Q, self.get_Q(state, (r, o)))
    # logging.info("Max Q: {}".format(max_Q))
    return max_Q

def get_average_Q(self, state: Nim):
    """Return average Q-value for state."""
    total_Q = 0
    for r, c in enumerate(state.rows):
        for o in range(1, c+1):
            total_Q += self.get_Q(state, (r, o))
    return total_Q / len(state.rows)

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

def get_action(self, state: Nim):
    """Return action based on epsilon-greedy policy."""
    if random.random() < self.epsilon:
        return random.choice(self.get_possible_actions(state))
    else:
        logging.info("Getting best action")
        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)
        return best_action

def register_state(self, state: Nim):
    # for each possible move in state, initialize random Q value
    for r, c in enumerate(state.rows):
        for o in range(1, c+1):
            if (hash_list(state.rows), (r, o)) not in self.Q:
                val = np.random.uniform(0, 0.01)
                # logging.info("Registering state: {} and action: {} to
                ↪  {}".format(state.rows, (r, o), val))
                self.set_Q(hash_list(state.rows), (r, o), val)
            else:
                logging.info("State already registered: {} and action:
                ↪  {}".format(state.rows, (r, o)))

```

```python
def update_Q(self, reward: int, game_over: bool):
    """Update Q-value for previous state and action."""

    if game_over:
        # self.set_Q(hash_list(self.previous_state.rows), self.previous_action, reward)
        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)))

    else:
    # if reward != -1:
        self.register_state(self.current_state)
        if self.previous_action is not None:
            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.gamma) * (self.get_max_Q(self.current_state) - self.get_Q(self.previous_state, self.previous_action)))
    # else:
    #     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)))

def print_best_action_for_each_state(self):
    for state in self.Q:
        logging.info("State: {}".format(state[0]))
        nim = Nim(5)
        nim.rows = unhash_list(state[0])
        logging.info("Best action: {}".format(self.choose_action(nim)))

def test_against_random(self, round, random_agent):
    wins = 0
    for i in range(rounds):
        nim = Nim(num_rows=5)
        player = 0
        while not nim.goal():
            if player == 0:
                move = self.choose_action(nim)
                # logging.info(f"Reinforcement move: Removed {move[1]} objects from row {move[0]}")
                nim.nimming_remove(*move)
            else:
                move = random_agent(nim)
                # logging.info(f"Random move {random_agent.num_moves}: Removed {move[1]} objects from row {move[0]}")
                nim.nimming_remove(*move)
            player = 1 - player

```

```python
            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
```

```python
        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)
```

```python
                    self.update_state(hash_list(self.current_state.rows), reward)
                    player = 1
                else:
                    action = opponent(self.current_state)
                    self.current_state.nimming_remove(*action)
                    player = 0

                if self.current_state.goal():
                    logging.info("Player {} wins!".format(1 - player))
                    break

            winner = 1 - player
            if winner == 0:
                agent_wins += 1
            # episodic learning
            self.learn()

            if episode % 1000 == 0:
                logging.info("Win rate: {}".format(agent_wins / (episode + 1)))
        if not training:
            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

5 Conclusion

Ok bye.