Report CI 22/23

Cristiano Serra 305960

Lab 1 Code

```
import heapq
import random
import logging
from random import seed, choice
from typing import Callable
import numpy as np
N = 100
SEED = 42


class PriorityQueue:
    """A basic Priority Queue with simple performance optimizations"""

    def __init__(self):
        self._data_heap = list()
        self._data_set = set()

    def __bool__(self):
        return bool(self._data_set)

    def __contains__(self, item):
        return item in self._data_set

    def push(self, item, p=None):
        assert item not in self, f"Duplicated element"
        if p is None:
            p = len(self._data_set)
        self._data_set.add(item)
        heapq.heappush(self._data_heap, (p, item))

    def pop(self):
        p, item = heapq.heappop(self._data_heap)
        self._data_set.remove(item)
        return item

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))
    ]
```

```python
PROBLEM = problem(N, SEED)

def goal_test(state):
    new = set()
    #Concatenation of all lists without duplicates, a state is a list of lists
(subset of problem)
    for list in state.data:
        new.add(list)

    test = [x for x in range(N)]

    return all(x in new for x in test) #Returns True if all elements of "test" are
contained in "new"

class State:
    def __init__(self, data: list):
        self._data = data.copy()
        #self._data.flags.writeable = False

    def __hash__(self):
        return hash(bytes(self._data))

    def __eq__(self, other):
        return bytes(self._data) == bytes(other._data)

    def __lt__(self, other):
        return bytes(self._data) < bytes(other._data)

    def __str__(self):
        return str(self._data)

    def __repr__(self):
        return repr(self._data)

    @property
    def data(self):
        return self._data

    def copy_data(self):
        return self._data.copy()

def possible_actions(state):
    #Return a possible action, in this case adding a list from the original problem
A(s)
    return [l for l in PROBLEM]

def result(state, action):
    #Return new state with action a performed R(s, a)
    return State(state.data + action)
```

```python
def search(
    initial_state: State,
    goal_test: Callable,
    parent_state: dict,
    state_cost: dict,
    priority_function: Callable,
    unit_cost: Callable,
):
    frontier = PriorityQueue()
    parent_state.clear()
    state_cost.clear()

    state = initial_state
    parent_state[state] = None
    state_cost[state] = 0

    while state is not None and not goal_test(state):
        for a in possible_actions(state):
            #print(f"Action: {a}")
            new_state = result(state, a)
            #print(f"New State: {new_state}")
            cost = unit_cost(a)
            if new_state not in state_cost and new_state not in frontier:
                parent_state[new_state] = state
                state_cost[new_state] = state_cost[state] + cost
                frontier.push(new_state, p=priority_function(new_state))
                logging.debug(f"Added new node to frontier
(cost={state_cost[new_state]})")
            elif new_state in frontier and state_cost[new_state] >
state_cost[state] + cost:
                old_cost = state_cost[new_state]
                parent_state[new_state] = state
                state_cost[new_state] = state_cost[state] + cost
                logging.debug(f"Updated node cost in frontier: {old_cost} ->
{state_cost[new_state]}")
        if frontier:
            state = frontier.pop()
        else:
            state = None

    path = list()
    s = state
    while s:
        path.append(s.copy_data())
        s = parent_state[s]

    logging.info(f"Found a solution in {len(path):,} steps; visited
{len(state_cost):,} states")
    return list(reversed(path))
```

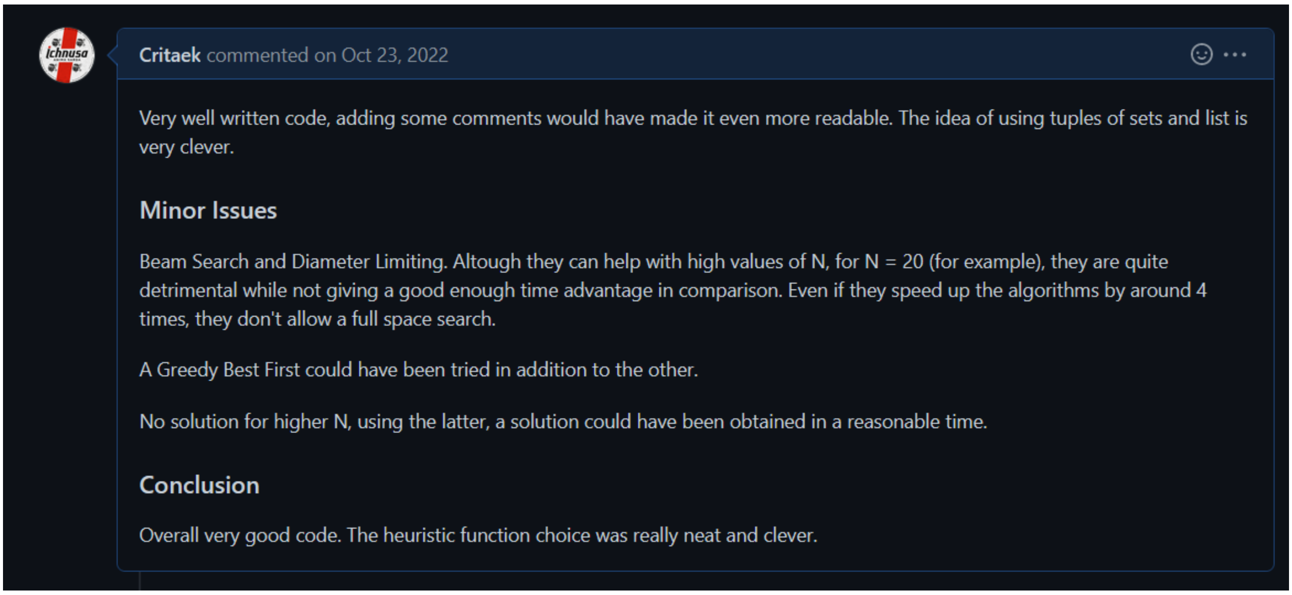
```python
parent_state = dict()
state_cost = dict()
INITIAL_STATE = State(list([]))

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

#print(PROBLEM)

final = search(
    INITIAL_STATE,
    goal_test=goal_test,
    parent_state=parent_state,
    state_cost=state_cost,
    priority_function=lambda s: len(state_cost),
    unit_cost=lambda a: len(a),
)

print(final)
```

Results:

N = 5 -> total length = 5 / nodes = ? / 1,175 states visited
N = 10 -> total length = 10 / nodes = ? / 160,922 states visited
N = 20 -> total length = 28 / nodes = ? / 2,807,347 states visited
N = 100 -> Memory error (16gb machine)
N = 500 -> Memory error

Lab 1 Reviews

Review aviable at: https://github.com/mistru97/CI_2022_s292623/issues/3



Critaek commented on Oct 23, 2022

Very well written code, adding some comments would have made it even more readable. The idea of using tuples of sets and list is very clever.

Minor Issues

Beam Search and Diameter Limiting. Altough they can help with high values of N, for N = 20 (for example), they are quite detrimental while not giving a good enough time advantage in comparison. Even if they speed up the algorithms by around 4 times, they don't allow a full space search.

A Greedy Best First could have been tried in addition to the other.

No solution for higher N, using the latter, a solution could have been obtained in a reasonable time.

Conclusion

Overall very good code. The heuristic function choice was really neat and clever.

Only one review done because I did not understand that I had to do two of them for each lab, but overall 6 reviews were done, one for the Lab 1, one for Lab 2, and four for Lab 3. I thought it was better to do some review for the last Lab instead of the older ones.

Lab 2 Code

Two versions of the algorithm were delivered, the first one is the result of some attempts, the second one is a more classical GA.

First Version Code

```
import random
from progress.bar import Bar

SEED = 42
POPULATION_SIZE = 100
MAX_GENERATIONS = 1000

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 random_gene():
        return random.choice([0,1])

class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    def cal_fitness(self):
        '''
        Fitness is the total length of the considered lists if the goal is reached
(all
        numbers between 0 and N-1 included) or maxint if not
        '''
        numbers = [x for x in range(N)]
        #contains only the contemplated lists
        lists = [PROBLEM[i] for i, x in enumerate(self.chromosome) if x == 1]
        new = set()
```

```
        for l in lists:
            new.update(l)

        a = all(x in new for x in numbers)
        if a:
            return sum([len(PROBLEM[i]) for i, x in enumerate(self.chromosome) if x
== 1])
        else:
            return len(PROBLEM) + 1

    @classmethod
    def generate_chromosome(self):
        return [random_gene() for _ in range(len(PROBLEM))]

    def mate(g1, g2):
        child_chromosome = []

        for gp1, gp2 in zip(g1.chromosome, g2.chromosome):
            prob = random.random()

            if prob < 0.45:
                child_chromosome.append(gp1)

            elif prob < 0.9:
                child_chromosome.append(gp2)

            else:
                child_chromosome.append(random_gene())

        return Individual(child_chromosome)

    def print_list(ind):
        lists = [PROBLEM[i] for i, x in enumerate(ind.chromosome) if x == 1]
        #print(lists)
        return lists

def offspring(N):
    generation = 0

    population = []

    #Generate initial population
    for _ in range(POPULATION_SIZE):
        rand_chromosome = Individual.generate_chromosome()
        population.append(Individual(rand_chromosome))

    with Bar("Processing", max = MAX_GENERATIONS) as bar:
        for _ in range(MAX_GENERATIONS):
            population = sorted(population, key = lambda x : x.fitness)
```

```
            if population[0].fitness == N:
                continue

            new_generation = []

            s = int((10*POPULATION_SIZE)/100)
            new_generation.extend(population[:s])

            s = int((90*POPULATION_SIZE)/100)
            for _ in range(s):
                parent1 = random.choice(population[:50])
                parent2 = random.choice(population[:50])
                child = Individual.mate(parent1, parent2)
                new_generation.append(child)

            population = new_generation

            generation += 1

            bar.next()

            #if generation % 100 == 0:
            #    print(f"N = {N} -> Generation: {generation}\tFitness:
{population[0].fitness}")

    print(f"N = {N} -> Generation: {generation}\tFitness: {population[0].fitness}")

if __name__ == "__main__":
    for N in [20]:
        PROBLEM = problem(N, SEED)
        #print(PROBLEM)
        offspring(N)
```

Second Version Code

```
import random
from progress.bar import Bar

POPULATION_SIZE = 100
MAX_GENERATIONS = 1000

def problem(N, seed=42):
    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 random_gene():
        return random.choice([0,1])

class Individual:
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    def cal_fitness(self):
        '''
        Fitness is the total length of the considered lists if the goal is reached
(all
        numbers between 0 and N-1 included) or maxint if not
        '''
        numbers = [x for x in range(N)]
        #contains only the contemplated lists
        lists = [PROBLEM[i] for i, x in enumerate(self.chromosome) if x == 1]
        new = set()

        for l in lists:
            new.update(l)

        a = all(x in new for x in numbers)
        if a:
            return sum([len(PROBLEM[i]) for i, x in enumerate(self.chromosome) if x
== 1])
        else:
            return len(PROBLEM) + 1

    @classmethod
    def generate_chromosome(cls):
        return [random_gene() for _ in range(len(PROBLEM))]

    def crossover(g1, g2):
```

```
        child_chromosome = []
        cut = int(len(g1.chromosome)/2)
        gp1 = g1.chromosome[0:cut]
        gp2 = g2.chromosome[cut:]
        child_chromosome.extend(gp1)
        child_chromosome.extend(gp2)

        return Individual(child_chromosome)

    def mutation(g1, g2):
        child_chromosome = []
        cut = int(len(g1.chromosome)/2)
        gp1 = g1.chromosome[0:cut]
        gp2 = g2.chromosome[cut:]
        child_chromosome.extend(gp1)
        child_chromosome.extend(gp2)
        #select a random gene and mutate it, from 1 to 0 or viceversa
        n = random.choice(range(len(g1.chromosome)))
        new_gene = 1 - child_chromosome[n]
        child_chromosome[n] = new_gene

        return Individual(child_chromosome)

    def print_list(ind):
        lists = [PROBLEM[i] for i, x in enumerate(ind.chromosome) if x == 1]
        return lists

def offspring(N):
    generation = 0

    population = []

    #Generate initial population
    for _ in range(POPULATION_SIZE):
        rand_chromosome = Individual.generate_chromosome()
        population.append(Individual(rand_chromosome))

    with Bar("Processing", max = MAX_GENERATIONS) as bar:
        for _ in range(MAX_GENERATIONS):
            population = sorted(population, key = lambda x : x.fitness)

            if population[0].fitness == N:
                continue

            new_generation = []

            s = int((10*POPULATION_SIZE)/100)
            new_generation.extend(population[:s])

            s = int((90*POPULATION_SIZE)/100)
```

```python
            for _ in range(s):
                parent1 = random.choice(population[:50])
                parent2 = random.choice(population[:50])
                #every 10 child, one will have a random mutation
                if _ % 10 == 0:
                    child = Individual.mutation(parent1, parent2)
                else:
                    child = Individual.crossover(parent1, parent2)

                new_generation.append(child)

            population = new_generation

            generation += 1

            bar.next()

            #if generation % 1 == 0:
            #    print(f" N = {N} -> Generation: {generation}\tFitness: 
{population[0].fitness}")

    population = sorted(population, key = lambda x : x.fitness)

    print(f"N = {N} -> Generation: {generation}\tFitness: {population[0].fitness}")

if __name__ == "__main__":
    SEED = 42
    for N in [20]:
        PROBLEM = problem(N, SEED)
        #print(PROBLEM)
        offspring(N)
```

The main difference is that in the first one the new generation is generated by keeping the best 10% of the initial population, and the remaining 90% is generated using the “mate” method, which creates a new individual based on a mix of crossover and mutation, 45% of the time it takes the gene from the first parent, the other 45% (0.45 < prob < 0.9) and the remaining 10% it mutates the gene.

In the second one, it still keeps the best 10%, and on the remaining 90% it will do a mutation every 10 individuals, while the others will be just a simple crossover between the parents.

First Version Results

N = 5 -> Generation: 3 String: [[0, 2], [3], [4], [1]] Fitness: 5
N = 10 -> Generation: 15 String: [[1, 3, 4, 9], [6], [8, 2, 7], [0, 5]] Fitness: 10
N = 20 -> Generation: 10001 String: [[18, 2, 15], [4, 5, 8, 13, 15, 16, 17, 19], [6, 9, 11, 12, 17], [2, 3, 7, 10, 14, 16], [0, 1, 2, 7]] Fitness: 26
N = 100 -> Generation: 10001 Fitness: 428
N = 500 -> Generation: 10001 Fitness: 74778 (population_size = 100)
N = 1000 -> Generation: 10001 Fitness: 357207 (population_size = 100)

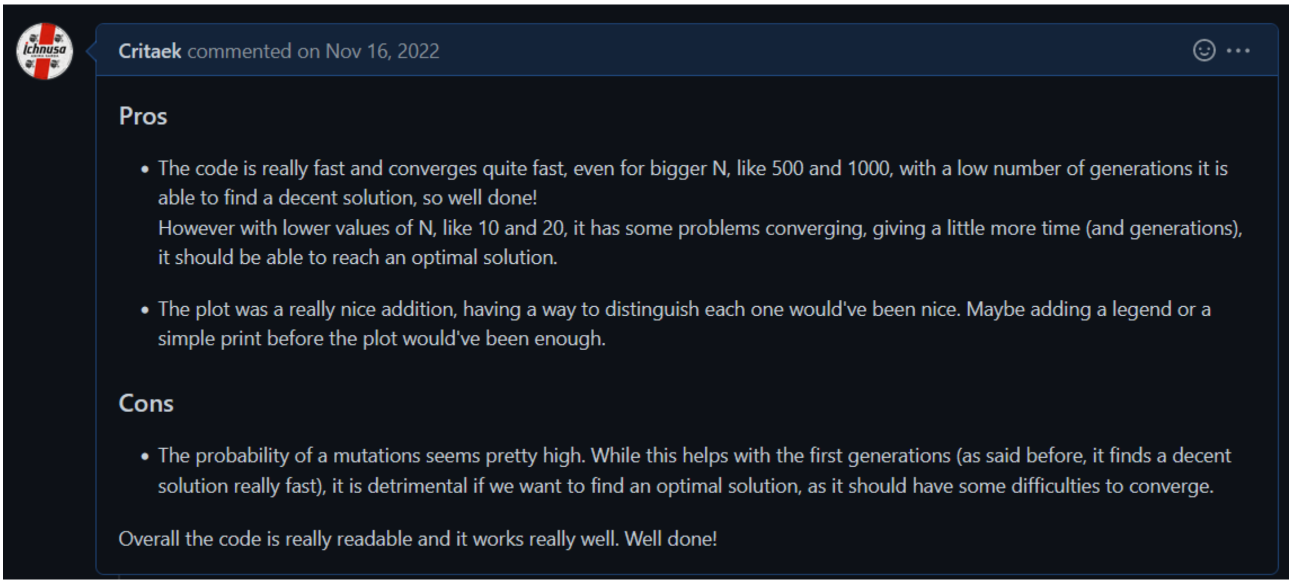
Second Version Results

N = 5 -> Generation: 11 Fitness: 5
N = 10 -> Generation: 10001 Fitness: 12
N = 20 -> Generation: 10001 Fitness: 35
N = 100 -> Generation: 10001 Fitness: 218
N = 500 -> Generation: 10001 Fitness: 1810
N = 1000 -> Generation: 10001 Fitness: 3620

As can be seen above, the second version is better for higher N, while it's worse for lower ones.

Lab 2 Reviews

Review avaiable at: https://github.com/mistru97/CI_2022_s292623/issues/7



Lab 3 Code

The Code is divided in 4 files, the Nim.py contains the Nim class, which is used to manage the game, the minmax.py contains the minmax algorithm to playthe game, RL.py contains the Reifnorcement Learning algorithm and the lab3.py contains a couple of fixed agents, a GA agent, and the main, which uses all of the above.

Nim.py

```python
from collections import namedtuple

Nimply = namedtuple("Nimply", "row, num_objects")

class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self._rows = [i * 2 + 1 for i in range(num_rows)]
        self._k = k

    # Example with k = 3, where k is the number of rows
    #   *
    #  ***
    # *****

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

    @property
    def rows(self) -> tuple:
        return tuple(self._rows)

    @property
    def k(self) -> int:
        return self._k

    # Make an action in the game (a ply is a move by one player)
    def nimming(self, ply: Nimply) -> None:
        row, num_objects = ply
        assert self._rows[row] >= num_objects
        assert self._k is None or num_objects <= self._k
        self._rows[row] -= num_objects

    # Given a Nim game, returns the possible moves in that state of the game
    def possible_moves(self):
        return [(r, o) for r, c in enumerate(self.rows) for o in range(1, c + 1) if
self.k is None or o <= self.k]
```

```
    # Returns if a game is in a terminal state or not
    def is_over(self):
        if sum(elements > 0 for elements in self.rows) == 0:
            return True
        else:
            return False

    def get_state_and_reward(self):
        return self, self.give_reward()

    def give_reward(self):
        # if at end give 0 reward
        # if not at end give -1 reward
        return -1 * int(not self.is_over())
```

minmax.py

```
from Nim import *
from copy import deepcopy

# In this case False is the player and True is the opponent
# player = True -> actual player
# player = False -> opponent
# We want to maximize True while minimizing False (the opponent)
def minmax(state: Nim, depth: int = 0, player: bool = True, max_depth: int = 4):
    # Get all the possible moves in the given state
    possible = state.possible_moves()

    # If the game is over
    if state.is_over() or not possible:
        if player == True:
            return (-1, None) # If True, and over or not possible, it means the
opponent won, negative feedback
        else:
            return (1, None) # If False, it means that player won, good!

    tried = []

    if depth == max_depth:
        return (0, None)

    # Each p is possible move (a ply)
    for m in possible:
        # For each ply, perform it, and call with the other player and so on
recursively
        new_state = deepcopy(state)
        new_state.nimming(m)

        value, move = minmax(new_state, depth+1, not player)
```

```
        tried.append((value, m))

    if player:
        return max(tried, key = lambda x: x[0])
    else:
        return min(tried, key = lambda x: x[0])

# Simple wrapper to take only the move and not the tuple
def minimax(state: Nim):
    return minmax(state, 0, True, 4)[1]
```

RL.py

```
import numpy as np
from copy import deepcopy
from Nim import Nim
import matplotlib.pyplot as plt
import random

class Agent(object):
    def __init__(self, alpha=0.15, random_factor=0.2):  # 80% explore, 20% exploit
        self.state_history = []  # state, reward
        self.alpha = alpha
        self.random_factor = random_factor
        self.G = {}
        # This two are used just to save and have a graphical plot of the "loss"
while learning
        self.moveHistory = []
        self.indices = []
        self.steps = -1

    def init_reward(self):
        return np.random.uniform(low=1.0, high=0.1)

    def choose_action(self, state: Nim, allowedMoves):
        maxG = -10e15
        next_move = None
        randomN = np.random.random()
        if randomN < self.random_factor:
            # if random number below random factor, choose random action
            next_move = random.choice(allowedMoves)
            state.nimming(next_move)
            if state not in self.G.keys():
                self.G[state.__str__()] = self.init_reward()

        else:
            # if exploiting, gather all possible actions and choose one with the
highest G (reward)
            for action in allowedMoves:
```

```
                new_state = deepcopy(state)
                new_state.nimming(action)

                if new_state not in self.G.keys():
                    self.G[new_state.__str__()] = self.init_reward()

                if self.G[new_state.__str__()] >= maxG:
                    next_move = action
                    maxG = self.G[new_state.__str__()]

        return next_move

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

    def learn(self):
        target = 0

        for prev, reward in reversed(self.state_history):
            self.G[prev.__str__()] = self.G[prev.__str__()] + self.alpha * (target
- self.G[prev.__str__()])
            target += reward

        self.state_history = []

        self.random_factor -= 10e-5  # decrease random factor each episode of play


    def play(self, state: Nim):
        nim = state
        # As in minmax, True is "us" while False is the opponent
        player = True

        for i in range(5000):
            self.steps += 1

            while not state.is_over():
                # state, _ = state.get_state_and_reward()  # get the current state
                state_copy = deepcopy(state)
                # choose an action (explore or exploit)
                action = self.choose_action(state_copy,
state_copy.possible_moves())
                state.nimming(action)  # update the maze according to the action
                state, reward = state.get_state_and_reward()  # get the new state
and reward
                # update the robot memory with state and reward
                self.update_state_history(state, reward)

            self.learn()  # robot should learn after every episode
            # get a history of number of steps taken to plot later
```

```
            if i % 50 == 0:
                print(f"{i}: {self.steps}")
                self.moveHistory.append(self.steps)
                self.indices.append(i)

            state = nim  # reinitialize the nim

        plt.semilogy(self.indices, self.moveHistory, "b")
        plt.show()
```

lab3.py

```
import logging
from Nim import *
import random
from typing import Callable
from copy import deepcopy
from itertools import accumulate
from operator import xor
from minmax import minimax
from RL import Agent

# Random Strategy, this is the same as the professor code, it takes a random number
# of sticks (objects) from a random row
def pure_random(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    num_objects = random.randint(1, state.rows[row])
    return Nimply(row, num_objects)

# Gabriele Strategy, also from the prof's code
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c
+ 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))


# Function that calculate the mathematical way of solving Nim
def nim_sum(state: Nim) -> int:
    # With *_, result we skip all the returned element apart from the last, which
will be store in result
    *_, result = accumulate(state.rows, xor)
    return result

def cook_status(state: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = state.possible_moves()
    cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
```

```
    cooked["shortest_row"] = min((x for x in enumerate(state.rows) if x[1] > 0),
key=lambda y: y[1])[0]
    cooked["longest_row"] = max((x for x in enumerate(state.rows)), key=lambda y:
y[1])[0]
    cooked["nim_sum"] = nim_sum(state)

    brute_force = list()
    for m in cooked["possible_moves"]:
        tmp = deepcopy(state)
        tmp.nimming(m)
        brute_force.append((m, nim_sum(tmp)))
    cooked["brute_force"] = brute_force

    return cooked

# Function that applies the matematical way of playing Nim
def optimal_strategy(state: Nim) -> Nimply:
    data = cook_status(state)
    return next((bf for bf in data["brute_force"] if bf[1] == 0),
random.choice(data["brute_force"]))[0]

# Function that make a strategy based on a genome, in this case a genome is a
probability,
# it can choose between a play or another
def strategy_to_evolve(genome: dict) -> Callable:
    def evolvable(state: Nim) -> Nimply:
        data = cook_status(state)

        if random.random() < genome["p"]:
            ply = Nimply(data["shortest_row"], random.randint(1,
state.rows[data["shortest_row"]]))
        else:
            ply = Nimply(data["longest_row"], random.randint(1,
state.rows[data["longest_row"]]))

        return ply

    return evolvable

# GLOBAL VARIABLES #
NUM_MATCHES = 10
NIM_SIZE = 5

# Function to perform a tournament with NUM_MATCHES matches, it returns the
fraction of games
# won by the strategy function (the first parameter), so if we pass as strategy the
optimal_strategy
# and as the opponent pure_random (for example), it will return 1.0
def evaluate(strategy: Callable, opponent: Callable) -> float:
        opponents = (strategy, opponent)
```

```python
        won = 0

        for m in range(NUM_MATCHES):
            logging.debug(m)
            nim = Nim(NIM_SIZE)
            player = 0
            while nim:
                ply = opponents[player](nim)
                nim.nimming(ply)
                player = 1 - player
            if player == 1:
                won += 1

        logging.debug(f" Player 0 won {won} time in {NUM_MATCHES} game,
corresponding to {won / NUM_MATCHES * 100} % of the games")

        return won / NUM_MATCHES


# In this individual, the lower the module, the best we can evolve the strategy
class Individual:
    def __init__(self, opponent: Callable):
        self.genome = {"p" : round(random.random(), 3)}
        self.module = 0.05
        self.opponent = opponent
        self.fitness = self.cal_fitness()

    def recal_fitness(self):
        self.fitness = self.cal_fitness()

        return self

    def cal_fitness(self):
        return evaluate(strategy_to_evolve(self.genome), self.opponent)

    def mutate(self):
        # In this case i want to have a random mutation in a direction
        # If -1, subtract "module" from "p"
        # If 1, add it
        direction = random.choice([-1,1])
        self.genome["p"] += direction * self.module

        # Update the fitness and return the updated Individual
        self.fitness = self.cal_fitness()

        return self

POPULATION_SIZE = 10
MAX_GENERATIONS = 10
```

```python
# Function that evolves the strategy_to_evolve playing against an opponent
# In this case "p" is the only parameter to optimize
def evolve(opponent: Callable):
    # Initialize the population
    # Each individual will have a random starting "p" with 3 decimal digits at the
start
    # and it will updated randomly at every generation
    population = []

    generation = 0

    for _ in range(POPULATION_SIZE):
        population.append(Individual(opponent))

    for _ in range(MAX_GENERATIONS):
        population = sorted(population, key = lambda x: x.fitness, reverse = True)

        new_generation = []

        # 10% of the best individuals will survive
        individuals_to_survive = int(0.1 * POPULATION_SIZE)
        for i in range(individuals_to_survive):
            new_generation.append(population[i].recal_fitness())

        # The other 90% will mutate
        mutants = population[individuals_to_survive:]
        for ind in mutants:
            new_generation.append(ind.mutate())

        population = new_generation
        generation += 1

        print(f"Generation -> {generation} Best found so far:
{population[0].fitness} with {population[0].genome}")

    population = sorted(population, key = lambda x: x.fitness, reverse = True)

    # Return the strategy with the best genome found
    return strategy_to_evolve(population[0].genome)

if __name__ == "__main__":
    logging.getLogger().setLevel(logging.DEBUG)

    # What the evaluate function returns is the WR of the first strategy passed, so
    # a value < 0.5 means the second one passed is better and a value > 0.5 means
    # the first passed is better

    # Match between a pure_random and gabriele
    #print( evaluate(pure_random, gabriele) )
    # gabriele wins!
```

```
    # We evolve a strategy by making it play against a hard coded rule
    # in this case, as gabriele is better than a random one, he will be the
opponent
    #evolved_strategy = evolve(gabriele)

    # Have a match between the evolved strategy and pure_random
    #print( evaluate(evolved_strategy, pure_random) )

    # Have a match between the evolved strategy and gabriele
    #print( evaluate(evolved_strategy, gabriele) )

    # Have a match between minmax and a random player
    #print( evaluate(gabriele, minimax) )

    agent = Agent()
    agent.play(Nim(2))
```

Lab 3 Reviews

In this lab, as said above (in the Lab 1 Reviews section), I did 4 reviews.

First review avaiable at: https://github.com/mistru97/CI_2022_s292623/issues/10

Critaek commented on Dec 19, 2022

Pros

Your expert strategy is quite impressive, winning even one in a while against nimsum is a really good result.
I liked how you wrote the evolutionary strategy, mainly the way the genome was constructed and used. Maybe adding a simple class for each individual would have made the code even more readable, but overall i had no problem in understanding how it works.
The minmax funtion is correct and the alpha-beta pruning does save a lot of time, mainly with bigger Nim values.

Cons

The biggest cons is the complete lack of the RL agent.
Maybe trying to evaluate againt different strategy could.ve been a good idea. Particularly when computing the win rate in the evolutionary strategy, having a stronger opponent (not too good, maybe something like gabriele), could've lead to even better results against the pure_random strategy, and overall would have made a stronger agent.

Final

The code is well written and I had no problem understanding.
The readme file is exaustive and beautifully written.
Overall well done!

Second review aviable at:
https://github.com/saccuz/Computational_Intelligence/issues/7

Critaek commented on Dec 20, 2022

Pros

A lot of fixed strategies, also the naming is really good, I instantly understood what they were doing even without reading all the code.
I liked the prograss bar in the evolutionary strategy (maybe adding a little comment on what tqdm does would've been a nice touch).
The way the ES works is quite smart, learning different opponents in a incremental way (by difficulty) is a nice idea, also the way the genome is used is pretty neat.
The README file in the task 3 was outstanding, the additon of graphs were a really nice touch and it was overall well explained and readable.
I'm not completely sure about the random moves added at the end after the depth limit, while it works pretty well, maybe just taking the best move at that depth would've been better, at least in my opinion.
The RL part was really well written, nothing much to add.

Cons

A little lack of comments in the task1-2, but overall not bad.
Minor (really minor) issue: In the readme files, the numbering is quite confusing, the first one is called part 1, but has both the 1st and 2nd part, so the last two are part 2 and 3 but they are actually referring to task 3 and 4 respectively, nothing too important, maybe having the same convention both for the task number and part would have been better.

Doubts

In this lines of code:

```
if results >= RESULT_TRESH:
        for i in range(len(genome)):
            if i in dna:
                genome[i] += EVOL_STEP
            else:
                genome[i] -= EVOL_STEP/2
    else:
        for i in range(len(genome)):
            if i in dna:
                genome[i] -= EVOL_STEP
            else:
                genome[i] += EVOL_STEP
```

Why in the first for we add EVOL_STEP or subtract EVOL_STEP/2, while in the second one we subtract EVOL_STEP and add EVOL_STEP, shouldn't we add EVOL_STEP/2 in the second one?

Final

I liked playing around with your code, apart from some lack of comments, the code was overall clean and organized. Good job!

Third review avaiable at:
https://github.com/b3xul/Computational_Intelligence/issues/5

Critaek commented on Dec 20, 2022

Pros

A lot of strategies implemented.
In the task 2, the Make_almost_optimal_strategy idea, even if not implemented/tried, it would've been really interesting and different compared to a more classical GA, i liked the idea a lot.
The minmax implementation was clean and fast, even with higher values of Nim.
The code was readable, a lot of comments, and a really good README file.
The tournament.tsv was really nice to read, it woudlve been a good addition to the README file.

Cons

In the task 2, the fact that you have the optimal strategy as one of the possible statregies to choose is a mistake. While interesting, and a confirmation that the algorithm works, it's trivial that the best possible strategy in the list is the optimal one.
Just deleting that would've been ok.
Lack of RL agent.

Final

I like how the code is structured. A lot of effort was put into it, even without the RL agent, the overall experience (reading and using the code) was really nice.

Fourth review avaiable at:
https://github.com/francescofiorella/computational_intelligence_2022_2023/issues/7

Critaek commented on Dec 20, 2022

Pros

I REALLY liked how the file was organized, highly readable.
The README file was also really exaustive and complete.
I liked how a lot of strategy were implemented, i believe you could have added "The new strategies" in the first part as a set of additional fixed rules.
The minmax implementation was clean and fast, nothing much to add, good job!
The RL agent part was simple and complete, easily understandable. The addition of graphs on each stage was really nice, and added a lot of context.

Cons

In the 2nd task, the way you are defining a population is a little particular, the way it is working is basically by finding the best strategy among a list of other strategies. In this kind of configuration i can't find the "evolutionary" part. A simple idea could've been to use a genome (maybe something like a list of probabilities?) to optimize and evolve, to then find the best strategy or combinations of those.
Even if the result would've been the same in the long run, i believe it lacks the primary idea of evolution.

Minor Cons

Maybe having a simple function to evaluate a strategy against another in a series of match, instead of having to write a for every time, would've been better. A simple wrapper would have been enough.

Final

Your code is really clean, readable, so does your README.
Good job!

Final Project

The approach used is a minmax algorithm with both alpha-beta pruning and a cache. When creating the player, it is possible to give the desired maximum depth. The two files I added are MyPlayer.py, which is my agent, and a Utils.py, the latter contains a couple of function to save or load a cache. It's then used when creating the MyPlayer, if a cache is possible, it will load it, if it's not it just creates an empty dictionary.

Also some other functions were added to the main.py, a tournament function, that given two players and a number of games, is able to compute the win rate of each player. A parallel_torunament which does the same thing but with parallelization (each thread runs a different game), and a build_cache function, that is able to build a cache for the minmax algorithm by running it and saving the cache with the function provided by the Utils class.

In the MyPlayer class, two other function were added, one can be used to save the cache after a game, and another one can be used to get the number of hits in the cache.

No changes were done in the Quarto.py or in the RandomPlayer.py.

Utils.py

```
import pickle

def save_cache(cache):
    with open("Cache/dict.pkl", "wb") as f:
        pickle.dump(cache, f)

def load_cache():
    try:
        with open("Cache/dict.pkl", "rb") as f:
            print("Loading cache...")
            cache = pickle.load(f)
            return cache
    except OSError as e:
        print("No cache found!")
        return None
```

Main.py

```python
import Quarto
from Player import RandomPlayer
import pickle
from joblib import Parallel, delayed
from MyPlayer import MyPlayer
import Utils

def play(player0, player1):
    game = Quarto.Quarto()
    n_0 = player0(game)
    n_1 = player1(game)
    game.set_players((n_0, n_1))
    return game.run()

def tournament(num, game):
    """
    Simple function that runs a tournament with num games
    between player0 and player1
    """

    win0 = 0
    win1 = 0
    for _ in range(num):
        game.reset()
        if _ % 1 == 0:
            print(_)
        winner = game.run()
        if winner == 0:
            win0 += 1
        if winner == 1:
            win1 += 1

    if win0 > win1:
        print(f"Player 0 won, with a winrate of: {(win0/(win0 + win1))*100}%")
    if win1 > win0:
        print(f"Player 1 won, with a winrate of: {(win1/(win0 + win1))*100}%")
    if win0 == win1:
        print(f"Draw")

def parallel_tournament(num, player0, player1):
    """
    Function that crate a torunament but with parallelization,
    each thread runes a game
    """
    results = Parallel(n_jobs = 4)(delayed(play)(player0, player1) for _ in
range(num))
    win0 = results.count(0)
    win1 = results.count(1)
```

```python
    if win0 > win1:
        print(f"Player 0 won, with a winrate of: {(win0/(win0 + win1))*100}%")
    if win1 > win0:
        print(f"Player 1 won, with a winrate of: {(win1/(win0 + win1))*100}%")
    if win0 == win1:
        print(f"Draw")

def build_cache():
    for _ in range(20):
        game = Quarto.Quarto()
        random_player = RandomPlayer(game)
        my_player = MyPlayer(game, 15)
        game.set_players((my_player, random_player))
        tournament(5, game)
        cache = my_player.get_cache()
        print(f"Cache size: {len(cache)}")
        print(f"In these game the cache hits were: {my_player.get_cache_hits()}")
        Utils.save_cache(cache)

if __name__ == "__main__":
    # build_cache()
    game = Quarto.Quarto()
    random_player = RandomPlayer(game)
    my_player = MyPlayer(game, 15)
    game.set_players((my_player, random_player))
    tournament(10, game)
    print(f"In these torunament the cache hits were: {my_player.get_cache_hits()}")
```

MyPlayer.py

```
import Quarto
import Utils
import copy
import numpy

load = Utils.load_cache()
cache = load if load else {}
if load != None:
    print(f"Loaded cache with len: {len(cache)}")

cache["hits"] = 0

MAX_DEPTH = 5

def get_empty_spaces_and_avaiable_pieces(board):
    empty_spaces = list()
    available_pieces = [x for x in range(16)]
    for x in range(4):
        for y in range(4):
            pos = board[y,x]
            if pos != -1:
                available_pieces.remove(pos)
            else:
                empty_spaces.append((x, y))
    return empty_spaces, available_pieces

def minmax(game: Quarto.Quarto, alpha = -1000, beta = 1000, flag = 0, depth = 0):
        # For every turn in quarto each player have to do two things, place a piece
        # and choose a piece to give to the opponent, the flag is used to do that,
        # it goes from 0 to 3, if it's 0 or 1 it means it's our turn, the first
time
        # so at 0 we are searching for a place, the second time it's always our
turn
        # but we are looking for the best piece (best for us) to give to the
opponent
        # the same happens with the other player, only that it's 2 or 3
        # When the complete turn is finished we can just check if we have 3 and go
back to 1
        searching_for_space = flag % 2 == 0
        its_us = flag < 2

        # If we reached the max depth or the game is finished, return 0 if it's
just finished,
        # -1 if it's us (that means we can't see any further, so it's better for
the opponent),
        # +1 if it's the opponent for the same reason before
        if depth == 0 or game.check_finished():
            return None, 0 if game.check_finished() else -1 if its_us else 1
```

```python
        # If someone won, give +1 is it's us, -1 if it's the opponent
        if game.check_winner() != -1:
            return None, 1 if its_us else -1

        board = game.get_board_status()
        hash = f"{numpy.array2string(board)} {flag}"
        if hash in cache.keys():
            cache["hits"] += 1
            return cache[hash]

        eval = list()
        empty_spaces, avaiable_pieces = get_empty_spaces_and_avaiable_pieces(board)

        for ply in empty_spaces if searching_for_space else avaiable_pieces:
            board = copy.deepcopy(game)
            if searching_for_space:
                board.place(ply[0], ply[1])
            else:
                board.select(ply)
            _, val = minmax(board, alpha, beta, flag=(flag+1)%4, depth=depth-1)
            eval.append((ply,val))
            if its_us:
                alpha = max(alpha, val)
            else:
                beta = min(beta, val)
            if beta <= alpha:
                break

        if its_us:
            val = max(eval, key = lambda k: k[1])
        else:
            val = min(eval, key = lambda k: k[1])
        cache[hash] = val
        return val

class MyPlayer(Quarto.Player):

    def __init__(self, quarto: Quarto.Quarto, max_depth = MAX_DEPTH) -> None:
        super().__init__(quarto)
        self.max_depth = max_depth

    def choose_piece(self) -> int:
        val = minmax(self.get_game(), flag=1, depth=self.max_depth)[0]
        return val

    def place_piece(self) -> tuple[int, int]:
        return minmax(self.get_game(), flag=0, depth=self.max_depth)[0]

    def get_cache(self):
```

```python
        return cache

    def get_cache_hits(self):
        return cache["hits"]
```

Thank you!