Computational Intelligence Report

Omar Andres Ormachea Hermoza
Student ID: 304811

Abstract

In this report, I will describe the work done throughout this course. It will include laboratory activities (code + peer reviews) and the final task: Building a program able to play the board game Quarto.

1 Introduction

This course tackles several computational paradigms in order to provide time/memory efficient solutions for complex problems, often resorting to heuristics and/or trial-and-error approaches. During this course there were 3 laboratory activities, consisting of implementing and understanding diverse solutions for the following problems:

1. Set covering - Search-based methods
2. Set covering - Genetic algorithms
3. Nim game
    (a) Fixed rules
    (b) Evolved rules
    (c) Minimax algorithm
    (d) Reinforcement Learning

At last, the final task for this course was to implement a player class able to play the game Quarto and hopefully achieve good performance against other players (random, human, etc.).

All the code and reviews shown on this report can be found on the following links (unless otherwise specified):

- Exam repository (private)
- Lab repository (public)

2 Laboratories

2.1 Lab 1: Set covering - Search-based methods

2.1.1 Problem

Given a number N and some lists of integers $P = (L_0, L_1, L_2, ..., L_n)$, determine, if possible, $S = (L_{s_0}, L_{s_1}, L_{s_2}, ..., L_{s_n})$ such that each number between 0 and $N-1$ appears in at least one list

$$\forall n \in [0, N-1]\ \exists i : n \in L_{s_i}$$

and that the total numbers of elements in all L_{s_i} is minimum. (Taken from Squillero's repository)

Our task was to implement Graph/Tree search in order to find the optimal solution problem for $N \in [5, 10, 20, 100, 500, 1000]$. The problem set (Python lists) are pseudo-randomly generated by a given function.

2.1.2 Approach

I implemented 3 Graph-search algorithms: Breadth-First, Depth-First and A*. The reason for the choice of Graph over Tree-based algorithms was that the Tree search algorithms, despite being more memory efficient than Graph-based (since they do not store the explored nodes set), are more computationally expensive for the fact that they may reach (explore) the same node multiple times. I considered that for this problem, time efficiency is more relevant than space efficiency.

At first, I implemented the brute-force approach for both Breadth-First and Depth-First. It seems that it works correctly, it manages to find the first solution at reasonable times. Then, it keeps searching for better solutions, eventually analyzing every possible state. I spent plenty of time looking for possible optimizations of the algorithm but in the end I realized that exploring all possible states for N=10 and beyond is unsolvable in acceptable time (and without crashing my PC, which I had to forcedly reboot more times in 4 days than in 2 years of use). Then, I refactored the code in order to just return the first solution found with the three proposed approaches.

Depth-First is expected to find a solution much faster than Breadth-First.

2.1.3 Code

gx_utils.py:

```python
# Copyright 2022 Giovanni Squillero <squillero@polito.it>
# https://github.com/squillero/computational-intelligence
# Free for personal or classroom use; see 'LICENSE.md' for details.

import heapq
from collections import Counter


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


class Multiset:
    """Multiset"""

    def __init__(self, init=None):
        self._data = Counter()
        if init:
            for item in init:
                self.add(item)

    def __contains__(self, item):
        return item in self._data and self._data[item] > 0

    def __getitem__(self, item):
        return self.count(item)
```

```python

    def __iter__(self):
        return (i for i in sorted(self._data.keys()) for _ in range(self._data[i]))

    def __len__(self):
        return sum(self._data.values())

    def __copy__(self):
        t = Multiset()
        t._data = self._data.copy()
        return t

    def __str__(self):
        return f"M{{{', '.join(repr(i) for i in self)}}}"

    def __repr__(self):
        return str(self)

    def __or__(self, other: "Multiset"):
        tmp = Multiset()
        for i in set(self._data.keys()) | set(other._data.keys()):
            tmp.add(i, cnt=max(self[i], other[i]))
        return tmp

    def __and__(self, other: "Multiset"):
        return self.intersection(other)

    def __add__(self, other: "Multiset"):
        return self.union(other)

    def __sub__(self, other: "Multiset"):
        tmp = Multiset(self)
        for i, n in other._data.items():
            tmp.remove(i, cnt=n)
        return tmp

    def __eq__(self, other: "Multiset"):
        return list(self) == list(other)

    def __le__(self, other: "Multiset"):
        for i, n in self._data.items():
            if other.count(i) < n:
                return False
        return True

    def __lt__(self, other: "Multiset"):
        return self <= other and not self == other

    def __ge__(self, other: "Multiset"):
        return other <= self

    def __gt__(self, other: "Multiset"):
        return other < self

    def add(self, item, *, cnt=1):
        assert cnt >= 0, "Can't add a negative number of elements"
        if cnt > 0:
            self._data[item] += cnt

    def remove(self, item, *, cnt=1):
        assert item in self, f"Item not in collection"
        self._data[item] -= cnt
        if self._data[item] <= 0:
            del self._data[item]

    def count(self, item):
        return self._data[item] if item in self._data else 0

    def union(self, other: "Multiset"):
        t = Multiset(self)
        for i in other._data.keys():
```

```
            t.add(i, cnt=other[i])
        return t

    def intersection(self, other: "Multiset"):
        t = Multiset()
        for i in self._data.keys():
            t.add(i, cnt=min(self[i], other[i]))
        return t
```

Imports, classes and search functions (solution.ipynb) (mostly based on professor's example):

```
import logging
import numpy as np
import random
from math import inf
from itertools import chain
from typing import Callable
from gx_utils import *

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

class State:
    def __init__(self, data: list):
        self._list = sorted(data.copy())
        self.set_covered = set(chain(*self._list))

    def __hash__(self):
        return hash(tuple(chain(*self._list)))

    def __eq__(self, other):
        return tuple(self.set_covered) == tuple(other.set_covered)

    def __contains__(self, other):
        return set(other) in self.set_covered

    def __le__(self, other):
        return self.set_covered <= other.set_covered

    def __lt__(self, other):
        return self.set_covered < other.set_covered

    def __str__(self):
        return str(chain(*self._list))

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

    def covers(self, other: list):
        return set(other) <= self.set_covered

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

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

def goal_test(state):
    return state.set_covered == goal

def possible_actions(state: State):
    return (l for l in all_lists if not state.covers(l))

def result(state, action):
    current_list = state.copy_data()
    current_list.append(action)
    return State(current_list)

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))
    ]
```

Generalized search algorithm for finding the optimal solution by brute force (doesn't finish execution until exploring each possible state) (solution.ipynb):

```
def search_min(
    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

    min_cost = inf
    min_state = None

    i = 0
    n_frontier = 0

    while state is not None:
        logging.debug(f'i = {i}')
        logging.debug(f'Current state -> {state.data}')

        if goal_test(state):
            logging.debug(f'Found a solution: {state.data}')
            if state_cost[state] < min_cost:
                logging.debug(f'Updating min cost -> {state_cost[state]}')
                min_cost = state_cost[state]
                min_state = state
                logging.info(f'New best solution: w = {min_cost} steps = {len(state.data)} (visited {i} nodes)')
        else:
            for a in possible_actions(state):
                new_state = result(state, a)
                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))
                    n_frontier += 1
                    logging.debug(f"Added new node ({n_frontier}) to frontier (cost={state_cost[new_state]}) -> {new_state.data}")

        if frontier:
            state = frontier.pop()
        else:
            state = None

        i += 1

    logging.debug(f'Total nodes in frontier: {n_frontier}')

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

    logging.info(f'Done in {i} iterations of the main loop')

    logging.debug('Min path followed:')
    logging.debug(list(enumerate(reversed(path))))
```

```

    return min_state
```

Generalized search algorithm returning the first solution found. This is the function that will be used for comparing the approaches (solution.ipynb):

```
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

    i = 0
    n_frontier = 0

    while state is not None and not goal_test(state):
        logging.debug(f'i = {i}')
        logging.debug(f'Current state -> {state.data}')
        for a in possible_actions(state):
            new_state = result(state, a)
            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))
                n_frontier += 1
                logging.debug(f"Added new node ({n_frontier}) to frontier (cost={state_cost[new_state]}) -> {new_state.data}")

        if frontier:
            state = frontier.pop()
        else:
            state = None

        i += 1

    logging.debug(f'Total nodes in frontier: {n_frontier}')

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

    logging.info(f'Visited {i:,} nodes')

    logging.debug('Path followed:')
    logging.debug(list(enumerate(reversed(path))))

    return state
```

Search deployment function (solution.ipynb). The difference between approaches is only in the **priority_function** selected:

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

for N in [5, 10, 20]:
    logging.info(f'N = {N}')
    goal = set(range(N))
    initial_state = State(list())

    all_lists = problem(N, seed=42)
```

```python

    all_lists = [list(t) for t in set(tuple(_) for _ in all_lists)] # Remove
    duplicates

    parent_state = dict()
    state_cost = dict()

    solution = search(
        initial_state,
        goal_test=goal_test,
        parent_state=parent_state,
        state_cost=state_cost,
        priority_function= # Difference is here
        unit_cost=lambda a: len(a),
    )

    logging.info(
        f"Found solution for N={N}: w={sum(len(_) for _ in solution.data)} (steps={len
    (solution.data)}) (bloat={(sum(len(_) for _ in solution.data)-N)/N*100:.0f}%)"
    )
```

Breadth-First:

```python
priority_function=lambda s: len(state_cost),
```

```
Output:
N = 5
Visited 48 nodes
Found solution for N=5: w=6 (steps=3) (bloat=20%)
N = 10
Visited 1,001 nodes
Found solution for N=10: w=11 (steps=3) (bloat=10%)
N = 20
Visited 6,587 nodes
Found solution for N=20: w=29 (steps=4) (bloat=45%)
```

Depth-First:

```python
priority_function=lambda s: -len(state_cost),
```

```
Output:
N = 5
Visited 4 nodes
Found solution for N=5: w=6 (steps=4) (bloat=20%)
N = 10
Visited 6 nodes
Found solution for N=10: w=16 (steps=6) (bloat=60%)
N = 20
Visited 8 nodes
Found solution for N=20: w=53 (steps=8) (bloat=165%)
N = 100
Visited 11 nodes
Found solution for N=100: w=339 (steps=11) (bloat=239%)
N = 500
Visited 18 nodes
Found solution for N=500: w=2679 (steps=18) (bloat=436%)
N = 1000
Visited 16 nodes
Found solution for N=1000: w=4880 (steps=16) (bloat=388%)
```

A*: Three priority functions defined:

```python
def h1(state: State):
    return len(list(chain(*state.data)))
```

```
Output:
N = 5
Visited 135 nodes
Found solution for N=5: w=5 (steps=3) (bloat=0%)
N = 10
Visited 40,000 nodes
Found solution for N=10: w=10 (steps=5) (bloat=0%)
N = 20
Visited 60,983 nodes
Found solution for N=20: w=23 (steps=5) (bloat=15%)
```

```python
def h2(state: State):
    c = Counter(list(chain(*state.data)))
    return c.total() - len(list(c))
```

```
Output:
N = 5
Visited 40 nodes
Found solution for N=5: w=5 (steps=3) (bloat=0%)
N = 10
Visited 975 nodes
Found solution for N=10: w=10 (steps=5) (bloat=0%)
N = 20
Visited 3,560 nodes
Found solution for N=20: w=23 (steps=5) (bloat=15%)
```

```python
def tup_priority_f(new_state: State):
    c = Counter(list(chain(*new_state.data)))
    return (c.total() - len(list(c)), -sum(c[e] == 1 for e in c))
```

```
Output:
N = 5
Visited 3 nodes
Found solution for N=5: w=5 (steps=3) (bloat=0%)
N = 10
Visited 4 nodes
Found solution for N=10: w=10 (steps=3) (bloat=0%)
N = 20
Visited 1,604 nodes
Found solution for N=20: w=23 (steps=5) (bloat=15%)
```

The first attempt was choosing **h1(.)** as the number of total elements of a given state, giving priority to short-length lists. This is sub-optimal and not admissible since shorter lists will require more nodes to be explored before reaching a full set coverage.

A second idea consists in a heuristic function **h2(.)** that gives priority directly considering the "bloat". For this purpose, a **Counter** object will used. Given a state, the function creates a **Counter** for it. Then, it will return the bloat. This approach yields good solutions but is quite slow. This **h2(.)** function may not be admissible (to be confirmed...). For N=100 and beyond, the execution never ends.

The third approach (completed after the deadline) was inspired by the professor's suggestion of exploiting hierarchical comparison of Python tuples, so that this last heuristic function returns a tuple, in order to consider more information when choosing the next node to explore. It is called **tup_priority_f**. It considers as a first element in the tuple the same as **h2(.)** (the total bloat) and the second element is the negative of the number of distinct elements in the new_state.

2.1.4 Comments

- The class **State** was modified in order to keep both a list of lists (**_list** attribute) and a set representing the set coverage of such state (**set_covered** attribute).
  - The method **covers()** was used for indicating whether a given list is a subset of the covered set of the state.
- In order to improve memory efficiency, after a state reaches the goal, it won't be explored for further nodes.
- A potential new node won't be considered an *available action* if the incoming list is a subset of the already covered set (this avoids repetition of lists since **all_lists** variable is never modified and it is scanned again at every iteration).
- The **__init__()** method stores the list of lists sorted such that another State with the same list of lists in different order is not considered nor stored in the frontier.
- Removing the list duplicates before deploying the search algorithm is useful since, even though a state won't consider a repeated list, different paths will eventually reach different instances of a duplicate list once.

2.1.5 Results

P_function	**N**	**w**	**Steps**	**Nodes visited**
Breadth	5	6	3	48
Depth	5	6	4	4
A* - h1	5	5	3	135
A* - h2	5	5	3	40
A* - tup	5	5	3	3
Breadth	10	11	3	1001
Depth	10	16	6	6
A* - h1	10	10	5	40000
A* - h2	10	10	5	975
A* - tup	10	10	3	4
Breadth	20	29	4	6587
Depth	20	53	8	8
A* - h1	20	23	5	60983
A* - h2	20	23	5	3560
A* - tup	20	23	5	1604

Table 1: Results (Lab 1) for N $\in [5, 10, 20]$

2.1.6 Peer reviews

I reviewed two classmates' code for lab 1.

1) Marco Masera:

oormacheah commented on Oct 23, 2022

Lab 1 review by Omar Ormachea

First of all, I want to say that I'm impressed that this code was made from scratch without using the given code snippets and even with a recursive approach for the DFS. Aside from that, as Luca said in his review, it is very difficult to follow the logic of the program without comments or a more explanatory README file.

After debugging I managed to understand a bit more how the programs work.

I followed a similar approach as your A* implementation, which as you know is very memory inefficient since all discovered nodes are being stored in memory at each node processing. The usage of a tuple as "priority" value for the heuristic function is very clever.

Some minor issues I could spot:

- On the class `List`:
  - `self.transformedSet = set(content)` is an unnecessary cast since the `content` parameter is always passed as a `set` already.
- On the class `List`:
  - I didn't quite understand what this line does and the reason behind its unusual format (I apologize if it's some advanced OOP style that I don't know):
    `list = list = [l.intersectTo_Copy(self.remainingElems) for index, l in enumerate(lists_) if index > skip]`
    Only thing I could say is that I would avoid the name "list" for a variable.

Overall, great job!

Marco-Masera commented on Oct 24, 2022 via email — Owner

Hi. Thanks for the kind words.
I definitely will add comments and a more explanatory readme next time!
About the two points:
-set(originalSet) creates a copy of the original set instead of a reference. It's not a cast but a function that returns a new set.
-Absolutely calling the variable "list" wasn't a great idea, I can see it can be confusing
-The second of code you quoted iterates on a list of sets and for each one calls the intersectTo_copy() function, which returns a new set. It basically builds the list of sets for the given node. The index<cut serves to avoid copying sets that won't be needed for the given node; this I should have put in the readme: it's basically a rule to avoid generating different nodes with different permutations of the same list of sets. If a node has N sets, its i child will surely use set i, and the branch sprouting from it cannot use any set j with j<i.

Il Dom 23 Ott 2022, 12:44 Omar Ormachea Hermoza ***@***.***>
ha scritto:
...

oormacheah commented on Oct 24, 2022 — Author

It's clear now, thanks. I used a slightly different approach for tackling the problem of avoiding permutations of same lists, but your approach makes total sense. And I apologize for my misunderstanding on the cast to set. Have a good one!

I have to say here that the code was very, very difficult to debug because it lacked comments and it was written from scratch (without re-using given functions), so as I wrote on the review I could only point out minor details. The code was already providing very good results as well. For these reasons I won't explicitly reference Marco's code.

2) Alessia Leclercq:

oormacheah commented on Oct 23, 2022

Lab 1 review by Omar Ormachea

I will be reviewing your final version of lab1.
I would like to start saying that the comments, the README and the code in general is very readable and simple (in a positive sense) so good job!

Issues

- As you probably already know, your selected heuristic function `h()` for A* is sub-optimal since it doesn't consider the repetitions of numbers (weight to minimize) in the current state nor the incoming list. Thus, the solution found is good but it analyzes too many "bloated" nodes before moving on with the actually optimal nodes.

Comments

- Sorting the list of lists (implemented with `frozenset` and `HashableArray` objects) when storing them in `ALL_STATES` variable is a very significant optimization for avoiding the same list of lists in a different order.
- The `HashableArray` class is a good idea to solve the un-hashability problem. A simpler way of doing it would be to cast every list and list of lists to tuple objects at the beginning (sorted as well, as you did). In this way every instance is hashable and immutable.
- As you wrote on your README, the function `bytes()` gives an error for N > 200 because it cannot take iterables that contain numbers that are > 256. A quick fix for that would be removing this function completely (the function `hash(self._data)` should be enough if you work with tuples as mentioned above).
- `goal_test()` builds the set covered by the given state inside its scope each time it is called. The way I did it was to compute and save the covered set inside the `State` object when created so that the `goal_test()` function just checks if it is equal to the already known solution. Your approach seems more efficient since it will only construct the covered set when analyzing (and testing) a node. Good job once again.

Minor issues

- The name `ALL_STATES` is not accurate, since the lists alone do not compose a state, but just potential elements of the states.
- The first import should be `from lab1_afterdeadline import *` and not `from lab1 import *`.

AlessiaLeclercq commented on Oct 23, 2022 Owner

Hi Omar, thank you for your review.
I'll try to run it without the use of bytes() and check whether it works properly!
:)

For understanding the review, I provide some code excerpts from Alessia's repository.

```python
def h(state: frozenset, N):
#heuristic function returns how many elements are missing to reach the solution
#work as goal_test except for the returned result
  element_set = set()
  for array_ in state:
    element_set.update(array_._data)
  return N-len(element_set)
```

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

Function above assumes self._data is a list of lists.

```python
def goal_test(state: frozenset, N):
#reads the data and adds the Hashablearrays values in the goal_set
#then checks whether the goal_set has lenght N (all values have been added)
  goal_set = set()
  for list_ in state:
    goal_set.update(list_._data)
  return len(goal_set) == N
```

2.2 Lab 2: Set covering - Genetic algorithms

2.2.1 Problem

Same as Lab 1.

2.2.2 Approach

A genetic algorithm was implemented starting by the example given in class (One-Max). I interpreted the validity of a solution wrongly: I assumed that it is okay for the genetic algorithm to "invent" solutions with lists that are not contained in the original problem set. This happens because, in my implementation, the cross-over and mutations may alter the internal lists as well, creating "new" elemental lists of integers. A simpler approach was to just work on a higher level with the existing lists of integers, without altering them, but altering their presence in a certain individual. The following key points characterize my solution:

- **Initial population**: The initial population was generated by sampling the original list of lists from lab1 **POPULATION_SIZE** times and taking a random amount of lists at each iteration. Each sample (subset) was casted to an **Individual** object.
- **Genome**: A list of lists (of unconstrained size) is a genome.
- **Fitness**: The measure for fitness is a tuple containing 3 elements (in order of priority):
  1. Number of distinct elements covered by the genome
  2. Bloat: Negative sum of the elements' multiplicity (if it isn't 1)
  3. Number of elements that appear only once in the genome
- **Cross-over**:
  - *Problem*: Two given genomes have no constraints about the size of the genome (i.e. number of lists nor their length)
  - *Idea*: Generate a random float from 0 to 1. It will correspond to a cut percentage and the cuts will be made proportionally. For example, assume we have g1 and g2 and the random number turns out 0.25. So, the new genome will be constituted of the first 25% of g1 and the last 75% of g2. I couldn't think of a more "fair" way to perform the cut.
- **Mutation**:
  - *Problem*: The selected element may turn into an invalid number for the problem (e.g. -1 if the number was originally 0 and it is subtracted by 1). This doesn't completely destroy the algorithm, but it may consider the corresponding genome "fitter" because it will add the -1 to the set covered, which is definitely wrong.
  - *Solution*: In these limit cases, the mutation is not random anymore. It is enforced (+1 or -1) to be such that the number doesn't fall out of range. It may not be the most fair approach.

The results can slightly vary by tweaking the parameters **POPULATION_SIZE**, **OFFSPRING_SIZE, NUM_GENERATIONS**, **TOURNAMENT_SIZE** and **MUTATION_RATE**. The algorithm is in general fast, though it provides sub-optimal results.

2.2.3 Code

GA_functions.py

```python
from itertools import chain
from collections import Counter
import random

def goal_test(genome, goal):
    return set(chain(*genome)) == goal

def problem(N, seed=None):
    random.seed(seed)
    return tuple(
        tuple(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 set_covering_fitness(genome):
    c = Counter(tuple(chain(*genome)))

    # Fitness will favor maximum values, so the number of covered elements should be a positive quantity and the bloat, negative.
    return (len(c), len(c) - c.total(), sum(c[e] == 1 for e in c))

def tournament(population, tournament_size=2):
    return max(random.sample(population, k=tournament_size), key=lambda i: i.fitness)

def cross_over(g1, g2):
    cut_percent = random.random() # For the proportional cut
    cut1 = int(cut_percent * len(g1))
    cut2 = int(cut_percent * len(g2))
    return g1[:cut1] + g2[cut2:]

def mutation(g, N):
    outer_point = random.randint(0, len(g) - 1) # Index on the outer list (genome)

    mut_list = g[outer_point] # List selected to be mutated from the genome

    inner_point = random.randint(0, len(mut_list) - 1) # Index of element to be mutated

    # Mutation of the element by adding or subtracting 1 to the randomly chosen element

    if mut_list[inner_point] == 0:
        # Force adding 1 if element is 0
        mutated_elem = mut_list[inner_point] + 1
    elif mut_list[inner_point] == (N - 1):
        # Force subtracting 1 if element is N - 1
        mutated_elem = mut_list[inner_point] - 1
    else:
        # If the value is not on the extrema, select either +1 or -1 (randomly)
        mutated_elem = mut_list[inner_point] + random.choice([-1, 1])

    modified_list = mut_list[:inner_point] + (mutated_elem,) + mut_list[inner_point + 1 :]

    return g[:outer_point] + (modified_list,) + g[outer_point + 1 :]
```

solution.py:

```python
import logging
import random
from collections import namedtuple

from GA_functions import *

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

N = 1000
```

```python

POPULATION_SIZE = 300
OFFSPRING_SIZE = 300
NUM_GENERATIONS = 1000
TOURNAMENT_SIZE = 2
MUTATION_RATE = 0.3

GOAL = set(range(N))

Individual = namedtuple("Individual", ["genome", "fitness"])

def main():
    list_of_lists = problem(N, seed=42) # Original problem generation

    list_of_lists = tuple(t for t in set(_ for _ in list_of_lists)) # Remove duplicate lists and cast to tuples

    # Initial population -> random selection and cast to Individuals (not a tournament)
    population = list()
    for _ in range(POPULATION_SIZE):
        subset_list = tuple(random.sample(list_of_lists, k=random.randint(1, len(list_of_lists)))) # Random subset of the lists of lists
        population.append(Individual(subset_list, set_covering_fitness(subset_list)))

    # Evolution

    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            if random.random() < MUTATION_RATE:
                p = tournament(population, tournament_size=TOURNAMENT_SIZE)
                o = mutation(p.genome, N)
            else:
                p1 = tournament(population, tournament_size=TOURNAMENT_SIZE)
                p2 = tournament(population, tournament_size=TOURNAMENT_SIZE)
                o = cross_over(p1.genome, p2.genome)
            f = set_covering_fitness(o)
            offspring.append(Individual(o, f))
        population += offspring
        population = sorted(population, key=lambda i: i.fitness, reverse=True)[:POPULATION_SIZE]

    for idx, i in enumerate(population):
        logging.info(f'individual {idx + 1} -> fitness: {i.fitness}, solved problem? {goal_test(i.genome, GOAL)}, w={sum(len(_) for _ in i.genome)}')

if __name__ == '__main__':
    main()
```

2.2.4 Results

These are the most relevant results:

- **N = 100**: w = 280 ∼ 350
- **N = 500**: w = 2200 ∼ 2500
- **N = 1000**: w = 6300 ∼ 6700

2.2.5 Peer reviews

I reviewed two classmates' code for lab 2:

1) Marco Masera:

oormacheah commented on Nov 13, 2022

Hi,

Overall I have to say that your code is well organized and readable, plus the comments and README file are very helpful for reviewing purposes. It is very well appreciated!

Getting deeper into the code, I have a couple things I'd like to point out. They are mostly about the notations and theoretical concepts used. There is nothing substantially "wrong" or improvable about your code that I could find.

Observations

- The representation you chose for the problem definitely constrains the "kind" of GA you can use. In your case, the `elementCovers` array makes it necessary to work with only feasible solutions since the beginning. I would consider this solution to be more of a *Hill Climbing* algorithm with Genetic tweaks.
- I think the terms `POPULATION_SIZE` and `TOURNAMENT_SIZE` are used inaccurately (not that this compromises the quality of your solution, it's just an observation). As far as I understood:
  - `POPULATION_SIZE` should be the number of individuals at the start of each new generation (in your solution, each new generation is consisting only of the offspring).
  - As for the `TOURNAMENT_SIZE`, it should represent the number of randomly chosen individuals for a "tournament" between them (e.g. for parent selection). In your solution, the parent selection is completely random so there are no "tournaments" happening.

Marco-Masera commented on Nov 13, 2022 via email Owner

Hi. Thanks for your review.
Regarding the misuse of the terminology you are completely right; that's because when I started writing the code I had in mind a different implementation, and as I changed it i did not rename the variables, which I should have done.
Especially the tournament number became simply the number of individual chosen for reproduction at each generation.

I'm not sure that restricting the possible genome space to feasible solutions makes it a Hill Climbing tho, individuals are still generated mixing the genomes of two others, with no overimposed "direction" of evolution.

Il Dom 13 Nov 2022, 22:26 Omar Ormachea Hermoza ***@***.***>
ha scritto:

...

oormacheah commented on Nov 13, 2022 Author

You are right, I got confused about the terminologies. Starting with a feasible solution is definitely a particular case of GA, but in principle it should be a GA.

Marco worked with a representation that allows only feasible solutions, which is not something I considered, so I provided a misleading feedback about the terminology. The **elementCovers** variable was initialized like this:

```
generated = [
        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))
    ]
    #Initialize the two global variables. Then we can forget about the generated lists
    , we don't need them anymore.
    listsCost = np.array([ len(s) for s in generated]) #Size of each list
    elementCovers = A=np.empty((N,),dtype=object) #Initialize elementCovers as an
    empty array of python objects (don't need np array for inner lists)
    for index, l in enumerate(generated): #Initialize the inner lists of elementCovers
     with the indexes
        for element in l:
            if (elementCovers[element] == None):
                elementCovers[element] = [index]
            else:
                elementCovers[element].append(index)
```

And the generation of the initial population is as following:

```
def get_random_individual():
    new_genome = np.empty((N,),dtype=np.int64)
    for target in range(0, N):
        new_genome[target] = elementCovers[target][random.randint(0, len(elementCovers
    [target])-1)]
    return Individual(new_genome)
```

This means that every individual starts off being a solution for the problem. It still provides a fitness measure that allows the evolution to be directed towards a better solution (in terms of bloat) after some generations. The mutations and cross-overs are random, as it is supposed to be for a Genetic Algorithm.

2) Giovanni Genna:

oormacheah commented on Nov 14, 2022

Hi,

I would like to start saying that your proposed solutions are clever and yield good results, as you probably already know :D. After some debugging it is not very difficult to catch the logic of the program. In terms of correctness of your solution I have honestly nothing to say. I could point out some personal observations:

- I would suggest to add some more comments here and there since the general intention of the algorithm is effectively explained, but some lines of code are quite long and it can get confusing when following step by step.
- In general, I think the handling of the `set` objects could be done in a **very slightly** more efficient (and readable) way. Just as an example, in the `add_list` function, you have these lines of code:

```
if not any(element == i for element in state[0]):
    state[0]= set(list(state[0]) + list(l))
    state[1]= set(list(state[1]) + [lists.index(l)])
    return
```

The following lines do the same, though they exploit the `set` object properties:

```
if i not in state[0]:
    state[0] |= set(l)
    state[1] |= set([lists.index(l)])
    return
```

Other than these I can't really find something substantially important to mention. Even the previous comment is a bit of a stretch from my side, I'll admit. Props to you for implementing even your personal fused approach, it is quite impressive!

2.3 Lab 3 - Nim game

2.3.1 Problem

Nim is a mathematical game of strategy in which two players take turns removing (or "nimming") objects from distinct heaps or piles. On each turn, a player must remove at least one object, and may remove at most K number of objects provided they all come from the same heap or pile. Depending on the version being played, the goal of the game is either to avoid taking the last object or to take the last object. (Taken from Wikipedia). For this course, the winner will be the player that **takes** the last object.

2.3.2 Approach

We saw in class that the optimal way of playing the game is well defined mathematically. It exploits a XOR operation throughout all the piles, taking the binary representation of the value of each pile. This is called the nim-sum. For our case: Assuming an agent plays first, if it plays every move ensuring that the nim-sum of the pile after its ply is equal to 0, it will always win. For N (number of piles) equal to 4, 8, and other values possibly multiple of 4, the optimal strategy does not ensure victory when playing first. In these cases, if the opponent plays optimally as well, it will win against the agent. Even one sub-optimal move from the opponent will grant back the secure victory to the agent.

Important note: I had a final exam for a course in France (I am doing an exchange there) around the last days of November + a project to deliver on the first week of December. For this reason, I couldn't complete lab 3 on time. I was able to finish it in January.

It was asked to implement 4 different agents able to play Nim.

1. **Fixed-rules**: My personal approach. This strategy is by no means attempting to compete with nim-sum. As a matter of fact, it is unable to beat it. It works as follows:

   (a) The ply will be done on the shortest active row.

   (b) Consider the number of active rows.

   (c) It this number is odd, the ply will be to either take all the remaining elements of the row or, if there is a K that doesn't allow to remove all, take just K (the maximum allowed) elements of this row.

   (d) If the number of active rows is even, the same logic holds, but the attempt will be to leave 1 element in the row, instead of removing all. If, again, the K doesn't allow this, just take K elements.

   This strategy performs better than **gabriele** and **pure_random**.

2. **Evolved rules**: I attempted to write a genetic algorithm for this problem. The general idea is that each individual carries in its genome a set of 5 probabilities that add up to 1. The probabilities determine how likely it is to select one of five fixed rules at each ply. The 5 fixed rules (in order) are: Remove 1, My fixed rule (from the previous task), Random, "Gabriele" strategy and the Optimal strategy. For example, a genome like this: $[0.1, 0.1, 0.1, 0.1, 0.6]$ is much more likely to select the last strategy (optimal) for each of its plies. It does not mean that it cannot select the other ones, though.

   - **Initial population**: The population is created by calling **POPULATION_SIZE** times the **random_genome** function. It generates 5 random decimal numbers and they are then normalized (for the sum to be 1).
   - **Genome**: The previously mentioned np-array consisting of a categorical distribution for 5 different values.
   - **Fitness**: Differently than a regular genetic algorithm, I did not define a fitness function. On the parent selection process, for one Individual to be considered "fitter" than the other, they have to go through a **Tournament**, which consists in randomly selecting 2 individuals to play **N_MATCHES** against each other. Each "match" actually consists on 2 matches, alternating the starting player on each of them. The actual number of Nim matches is then **2 * N_MATCHES** on each parent selection. After each match, the winner's win count is increased. The selected parent is the individual that won the most single matches.
   - **Cross-over**: If the selected individuals share on their genome the maximum value gene, it is kept and increased by a random factor (decimal value from 0 to 1). The remaining genes of the new genome are randomly selected from any of the two parents. If the individuals do not have a common highest gene, the new genome selects randomly from one of the parents for each of the genes. In both cases, the last step is a normalization of the array.
   - **Mutation**: Given a genome, a gene is selected randomly and it is increased or decreased (again, randomly) by a random decimal factor from 0 to 1.

   By altering the parameters, it can be seen that most individuals end up containing a gene almost equal to 1 (and all the remaining ones close to 0), meaning that the agent would just resort to one single strategy throughout a whole game of Nim. The **N_MATCHES** is very important. If it is low, parent selections will be mostly random, as in the beginning all the gene probabilities are roughly the same. A few number of victories is not sufficient to suggest that the actual better strategy was chosen enough times to prove responsible for the tournament victory. Unfortunately, the individuals don't always converge to the optimal strategy. This may be due to the randomicity of the initial population and the selection of the first tournament winners, which then may cause the growth of a sub-optimal gene. Eventually, at random as well, the genes that play the optimal strategy the most may be lost due to not having been selected for a tournament at all.

3. **Minimax**: I adapted the Minimax algorithm with depth limit and Alpha-Beta pruning from this source: Real Python. It successfully plays the optimal move at each turn, though it is quite slow. The game tree grows very fast, reason why it is important to set a depth limit for the game tree. In order to take some advantage from long term effects, a possible improvement would be to implement Monte Carlo Tree Search (MCTS) once the depth limit is reached. This was not done for this laboratory but will be implemented for the final task.

4. **Reinforcement Learning**: This implementation was inspired by the given RL maze example and github.com/Luigian/Nim. It is a Q-learning approach where at each state the agent will get a set of Actions and apply the $Q(s, a)$ function that will allow to get the value (reward) of a certain action from a certain state. In this implementation, the Q "function" is a Python dictionary that stores the Q value for all the state-action pairs. All the Q values are initialized as 0.

   The training works as following: Our agent plays against a Random player **n_episodes** times, allowing to learn the best moves while expecting various levels of opponents. In this way, the agent is able to learn how to beat even an optimal opponent consistently. If the agent was trained only against an optimal opponent, it would learn how to beat it but would not have any experience that it can use against other opponent strategies. This happens because a different player will most likely reach states that the optimal player never reaches, and the corresponding Q-values would be untouched since initialization.

   During the game, the agent has a parameter that accounts for the exploration vs. exploitation dilemma. If it is high, the Agent will be more prone to ignore the current Q-values and choose randomly its next action (exploration). If it is low, the Agent may pick the action corresponding to the maximum expected reward (exploitation). For training purposes, the results suggest that keeping a high exploration (parameter close to 1) is key to achieving good results. When evaluating the Agent, the parameter should be kept low (even 0) in order to exploit the acquired knowledge instead.

   When each Nim game ends, there is the **learning** phase, consisting of updating every Q-value corresponding to the actions that the agent took during that game. The end result (victory or loss) determines the value of the reward. For the update of the Q-table, there is the **learning rate** that accounts for how much of the "old" information should be preserved with respect to the "new" information, and the **discount factor** will determine how much we should consider the long-term reward with respect to the immediate reward. Keeping the highest value possible for the discount factor (1) gives the best results. It makes sense, since the only state that provides a real reward is a terminal state, when the game has ended, so it should have a significant impact on this computation for each of the previous states.

2.3.3 Code

nim.py: Taken from professor's repository + some additions in order to cover certain agents' missing functionalities. Includes also functions for simulating single matches and evaluating agents by playing multiple times.

```
import logging
from typing import Callable
from operator import xor
from copy import deepcopy
from itertools import accumulate, product
from collections import namedtuple


Nimply = namedtuple("Nimply", ['row', 'num_objects'])
```

```python

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

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

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

    def __hash__(self):
        return hash(tuple(self._rows))

    def __eq__(self, other):
        return (self.rows) == (other)

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

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

    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
        self._rows.sort()

    def possible_states(self):
        poss_val_per_row = [list(range(row_val + 1)) for row_val in self.rows]
        return tuple(set([tuple(sorted(t)) for t in product(*poss_val_per_row)]))

    def possible_moves(self):
        return [
            Nimply(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
        ]

    def construct_allowed_states(self):
        # create a dictionary of allowed state transitions from any board combination -> To optimize, consider equivalent game states
        # with a sorted tuple object
        allowed_states = {}
        for possible_state in self.possible_states():
            # iterate through all possible states, equivalent game states have been removed
            allowed_states[possible_state] = possible_moves_external(possible_state, self.k)
        self.allowed_states = allowed_states

    def is_game_over(self):
        # 'self' object is boolean-evaluated as sum(self._rows) > 0
        return not self

    def get_reward(self, winner=None):
        if winner is None:
            return 0
        elif winner == True:
            return 1
        elif winner == False:
            return -1

def possible_moves_external(rows_state: tuple, k: int=None):
    return [
```

```python
            Nimply(r, o) for r, c in enumerate(rows_state) for o in range(1, c + 1) if k is None or o <= k
        ]

def nimming_new_obj(state: Nim, ply: Nimply) -> Nim:
    state_copy = deepcopy(state)
    state_copy.nimming(ply)
    return state_copy

def nim_sum(state: Nim) -> int:
    *_, result = accumulate(state.rows, xor)
    return result

def cook_status(state: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = [
        Nimply(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state.k is None or o <= state.k
        # 'c' is total number of elements per row, 'o' is a number of objects to grab if it's below 'k'
    ]
    cooked["active_rows_number"] = sum(o > 0 for o in state.rows)
    cooked["sorted_rows"] = sorted(enumerate(state.rows), key=lambda y: y[1]) # My addition
    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) # Apply the ply (r, o) -> remove 'o' objects from row 'r' of the current state
        brute_force.append((m, nim_sum(tmp)))
    cooked["brute_force"] = brute_force

    return cooked

def single_match(strategy1, strategy2, nim_size, k=None):

    strategy = (strategy1, strategy2)

    nim = Nim(nim_size, k)
    logging.debug(f"status: Initial board  -> {nim}")
    player = 0 # Initial player
    while nim:
        ply = strategy[player](nim)
        nim.nimming(ply)
        logging.debug(f"status: After player {player} -> {nim}")
        player = 1 - player
    winner = 1 - player
    logging.info(f"status: Player {winner} won!")
    return winner

def evaluate(strategy: Callable, reference_strategy: Callable, NUM_MATCHES: int, NIM_SIZE: int, k=None, RL=False) -> float:
    '''
    Evaluate multiple games against a given strategy (usually nim-sum), your proposed strategy moves first
    '''
    logging.info(f"Evaluating over {NUM_MATCHES} matches on a board of {NIM_SIZE} rows...")

    opponent = (strategy, reference_strategy)
    won = 0

    for m in range(NUM_MATCHES):
        nim = Nim(NIM_SIZE, k, RL)
        player = 0 # Setting the passed strategy to perform the first move
        while nim:
```

```
            ply = opponent[player](nim)
            nim.nimming(ply)
            player = 1 - player

        # Exiting the loop, a player has won and it is stored in 'player'

        if player == 1:
            won += 1
    return won / NUM_MATCHES
```

strategies.py: All the strategies' (except Minimax and RL) wrappers.

```
from nim import *
import random
import numpy as np

strategies_str = [
    'remove1',
    'my_fixed_rule',
    'pure_random',
    'gabriele',
    'nim_sum',
]

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.k if state.k is not None and state.rows[row] > state.k else state.rows[row])
    return Nimply(row, num_objects)

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])))

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]
    # Iterator may be exhausted if no possible move gives 0 nim-sum, so that you would pick at random

def grab_one(state: Nim) -> Nimply:
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    return Nimply(row, 1)

# Task 3.1 - Fixed rules
def my_fixed_strategy(state: Nim) -> Nimply:
    data = cook_status(state)
    if all(r <= 1 for r in state.rows):
        return (random.choice(data["brute_force"]))[0]
    next_active_row = next(r for r, o in data["sorted_rows"] if o > 1)

    if data["active_rows_number"] % 2 == 0:
        # If number of rows is even, take all except 1 or take as many as possible from the shortest row
        # (bigger than 1)
        if state.k is None or state.rows[next_active_row] <= state.k + 1:
            return Nimply(next_active_row, state.rows[next_active_row] - 1) # Remove all except one
    else:
        # If number of rows is odd, try to take all or as many as possible from the shortest row
        if state.k is None or state.rows[next_active_row] <= state.k:
            return Nimply(next_active_row, state.rows[next_active_row]) # Remove all except one

    # If there is a k and you got more than k + 1 or k
    return Nimply(next_active_row, state.k) # Subtract the max k allowed

```

```python
# Task 3.2 - Evolvable strategy
def make_strategy(genome: np.ndarray) -> Callable:

    def evolvable(state: Nim) -> Nimply:
        chosen_strat = np.random.choice(strategies_str, 1, p=genome)[0]

        if chosen_strat == 'remove1':
            ply = grab_one(state)
        elif chosen_strat == 'my_fixed_rule':
            ply = my_fixed_strategy(state)
        elif chosen_strat == 'pure_random':
            ply = pure_random(state)
        elif chosen_strat == 'gabriele':
            ply = gabriele(state)
        elif chosen_strat == 'nim_sum':
            ply = optimal_strategy(state)

        return ply

    return evolvable
```

evolving_agent.py

```python
import random
import logging
from nim import *
from strategies import *
import numpy as np

NIM_SIZE = 4
POPULATION_SIZE = 50
OFFSPRING_SIZE = 30
NUM_GENERATIONS = 100
TOURNAMENT_SIZE = 2
N_MATCHES = 50
MUTATION_RATE = 0.3

Individual = namedtuple("Individual", ["genome"])

def random_genome(genome_length):
    probs = np.array([random.random() for _ in range(genome_length)])
    tot = probs.sum()
    normalized = probs / tot
    return normalized

def tournament(population, tournament_size=2):
    selected_individuals = random.sample(population, k=2)

    # The genomes will be put to test playing first and second for N_MATCHES times
    win_count = [0, 0]
    player0 = make_strategy(selected_individuals[0].genome)
    player1 = make_strategy(selected_individuals[1].genome)

    for i in range(N_MATCHES):

        # Match 1
        winner = single_match(player0, player1, NIM_SIZE)
        win_count[winner] += 1

        # Match 2 (inverted order)
        winner = single_match(player1, player0, NIM_SIZE)
        win_count[winner] += 1

    top_g = max(enumerate(win_count), key=lambda y: y[1])[0]
    return selected_individuals[top_g]

def cross_over(g1, g2):
    # Particular cross-over
    new_gene = np.empty(5)

    highest1 = max(enumerate(list(g1)), key=lambda x: x[1])[0]
```

```python
    highest2 = max(enumerate(list(g1)), key=lambda x: x[1])[0]

    if highest1 == highest2:
        # If both share the maximum gene, the maximum is chosen and it is increased at the expense of the others
        current_val1 = g1[highest1]
        current_val2 = g2[highest1]

        max_val = max(current_val1, current_val2)

        added_p = random.random() * max_val
        new_max = max_val + added_p
        new_gene[highest1] = new_max

        # new_gene[highest1] = max_val

        for i in range(len(new_gene)):
            if i != highest1:
                random.seed()
                new_gene[i] = (g1[i], g2[i])[random.choice([0, 1])]

    else:
        for i in range(len(new_gene)):
            random.seed()
            new_gene[i] = (g1[i], g2[i])[random.choice([0, 1])]

    norm_gene = np.array([i / new_gene.sum() for i in new_gene])
    return norm_gene


def mutation(g):
    selected = random.choice([[idx, ge] for idx, ge in enumerate(g)]).copy()
    g_copy = g.copy()

    p = random.random()
    if random.random() < 0.5:
        selected[1] -= selected[1] * p
    else:
        selected[1] += selected[1] * p

    g_copy[selected[0]] = selected[1]
    norm_gene = np.array([i / sum(g_copy) for i in g_copy])

    return norm_gene

# Genetic algorithm

def evolution():
    # Initial population
    population = list()
    for _ in range(POPULATION_SIZE):
        new_genome = random_genome(5)
        population.append(Individual(new_genome))

    # Evolution
    for g in range(NUM_GENERATIONS):
        offspring = list()
        for i in range(OFFSPRING_SIZE):
            if random.random() < MUTATION_RATE:
                p = tournament(population, tournament_size=TOURNAMENT_SIZE)
                o = mutation(p.genome)
            else:
                p1 = tournament(population, tournament_size=TOURNAMENT_SIZE)
                p2 = tournament(population, tournament_size=TOURNAMENT_SIZE)
                o = cross_over(p1.genome, p2.genome)
            offspring.append(Individual(o))
        population += offspring
        population = population[-1::-1][:POPULATION_SIZE]
        random.shuffle(population)

    for idx, i in enumerate(population):
```

```python
        print(f'individual {idx + 1} -> genome: {[round(n, 3) for n in list(i.genome)]}')

if __name__ == '__main__':
    evolution()
```

min_max_agent.py

```python
from nim import *
from functools import cache

def possible_moves(state: Nim):
    # Return a generator
    return (
        Nimply(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1) if state.k is None or o <= state.k
    )

@cache
def minimax(state: Nim, maximizingPlayer: bool, depth=None, alpha=-1, beta=1):
    if depth == 0 or not state:
        return 1 if not maximizingPlayer and not state else -1

    scores = []
    for child_ply in possible_moves(state):
        scores.append(
            score := minimax(nimming_new_obj(state, child_ply), not maximizingPlayer,
                depth - 1 if depth is not None else None, alpha, beta)
        )
        if maximizingPlayer:
            alpha = max(alpha, score)
        else:
            beta = min(beta, score)
        if beta <= alpha: # In Nim, the case of beta < alpha won't happen (both are either +1 or -1)
            break

    return (max if maximizingPlayer else min)(scores)

def minimax_strategy(depth=None, alpha=-1, beta=1) -> Callable:

    def best_possible_move_minimax(state: Nim) -> Nimply:
        return max(
            (minimax(nimming_new_obj(state, child_ply), False, depth=depth, alpha=alpha, beta=beta), child_ply)
            for child_ply in possible_moves(state)
        )[1] # Returns only the ply

    return best_possible_move_minimax
```

RL_agent.py

```python
import numpy as np
import random
from nim import *

class Agent(object):
    def __init__(self, state: Nim, alpha=0.15, random_factor=0.2, discount=0.4):
        self.state_action_history = []  # [(state, reward)]
        self.alpha = alpha
        self.random_factor = random_factor
        self.discount_factor = discount
        self.Q = {}
        self.init_reward(state)

    def init_reward(self, state: Nim):
        # Initialize reward for state, action pairs (Q-table)
        for s in state.possible_states():
            self.Q[s] = {}
            for m in possible_moves_external(s, state.k):
```

```python
                self.Q[s][m] = 0 # np.random.uniform(low=0.1, high=1.0)

    def choose_action(self, state, allowedMoves, evaluation=False):
        maxQ = -10e15
        next_move = None
        randomN = random.random()
        if randomN < self.random_factor and not evaluation:
            # if random number below random factor, choose random action
            next_move = random.choice(allowedMoves)
        else:
            # if exploiting, gather all possible actions and choose one with the highest Q value (reward)
            for action in allowedMoves:
                # new_state = nimming_new_obj(state, action)
                # print(new_state)
                # print(self.Q[new_state])
                if self.Q[state][action] >= maxQ:
                    next_move = action
                    maxQ = self.Q[state][action]

        return next_move

    def update_state_history(self, state, action, reward):
        self.state_action_history.append((state, action, reward))

    def change_state_history(self, state, action, reward):
        self.state_action_history[-1] = (state, action, reward)

    def learn(self):
        # target = 1
        # exit()
        i = -1
        for i in range(len(self.state_action_history) - 1):
            st, act, reward = self.state_action_history[i]
            new_st, _, _ = self.state_action_history[i + 1]
            self.Q[st][act] += self.alpha * (reward + self.discount_factor * max(self.Q[new_st].values()) - self.Q[st][act])
            # target += reward

        # Add the missing part of the puzzle
        i += 1
        st, act, reward = self.state_action_history[i]
        self.Q[st][act] += self.alpha * (reward - self.Q[st][act])

        self.state_action_history = []

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

    def ply(self, state: Nim) -> Nimply:
        return self.choose_action(state, state.allowed_states[state.rows], evaluation=True)
```

RL_main.py

```python
from nim import Nim
from RL_agent import Agent
from strategies import *
import matplotlib.pyplot as plt
import shelve

NIM_SIZE = 4
k = None
alpha = 0.2 # Learning rate
exploit_vs_explore = 0.8 # High exploration is key for achieving good results
discount_factor = 1 # Also, seems that considering the future reward only is better

training_opponent = pure_random
test_opponent = optimal_strategy
n_episodes = 3000
n_test_matches = 100

```

```python
stepSize = 50 # For status printing only

saveQtable = True # False for loading the Q-table
Q_table_path = './data/Q_table_x.data'

def train(NIM_SIZE, k, alpha, exploit_vs_explore, n_episodes, printStatus=False, printQ=False):
    state = Nim(NIM_SIZE, k, RL=True)
    bot = Agent(state, alpha=alpha, random_factor=exploit_vs_explore, discount=discount_factor)
    indices = []
    action = None
    win_count = 0

    for i in range(n_episodes):
        player = 0 # Setting starting player as the number 0 (default)
        bot_idx = random.choice((0, 1)) # Set the bot to play first or second randomly

        last_state = None
        last_action = None

        # Game loop
        while True:
            # state, _ = nim.get_state_and_reward(agent_playing=True)  # get the current state
            # choose an action (explore or exploit)
            if player == bot_idx:
                action = bot.choose_action(state, state.allowed_states[state.rows])

                # Save last state and action only if the bot is playing
                last_state = deepcopy(state)
                last_action = deepcopy(action)

                state.nimming(action)  # update the nim according to the action
            else:
                opponent_ply = training_opponent(state)
                state.nimming(opponent_ply)

            if state.is_game_over():
                if player == bot_idx:
                    # Bot won
                    reward = state.get_reward(winner=True)
                    bot.update_state_history(last_state.rows, action, reward)
                    win_count += 1
                else:
                    reward = last_state.get_reward(winner=False)
                    bot.change_state_history(last_state.rows, last_action, reward)
                break
            else:
                # If the game is not over, give 0 rewards
                if player == bot_idx:
                    reward = state.get_reward(winner=None)
                    bot.update_state_history(last_state.rows, action, reward)
            player = 1 - player

        bot.learn()  # robot should learn after every episode

        # This print is used to keep track of the training and the win count (not very relevant)
        if printStatus:
            if i % stepSize == 0:
                print(f"{i}: Win count: {win_count}")
                # indices.append(i)
            state = Nim(NIM_SIZE, k, RL=True)  # reinitialize the board

    if printQ:
        for k in sorted(bot.Q.keys()):
            print(k)
            for k1 in bot.Q[k]:
                print(k1, bot.Q[k][k1])
            print('\n')
```

```

    if saveQtable:
        # Save table
        pass

    return bot

# plt.semilogy(indices, moveHistory, "b")
# plt.show()

if __name__ == '__main__':
    if saveQtable:
        llamabot = train(NIM_SIZE, k, alpha, exploit_vs_explore, n_episodes)
    else:
        # Load Q-table from Q_table_path
        pass

    win_rate = evaluate(llamabot.ply, test_opponent, n_test_matches, NIM_SIZE, k, True
    )
    print(f'Win rate against {test_opponent.__name__} (Bot playing FIRST): {round(
    win_rate * 100, 3)} %')
    win_rate = evaluate(test_opponent, llamabot.ply, n_test_matches, NIM_SIZE, k, True
    )
    print(f'Win rate against {test_opponent.__name__} (Bot playing SECOND): {round(
    win_rate * 100, 3)} %')
```

3 Final task

3.1 Introduction

Quarto is a two-player strategy game where the objective is to place pieces on a 4x4 grid such that the pieces of a specific attribute match in a row, column, or diagonal. The game is played with 16 unique pieces that each have four different attributes, such as size, shape, color, and texture. (Written by chatGPT)

The objective of this task is to write a **Player** sub-class able to play Quarto with some of the paradigms studied during this course. As a side note, I implemented a **HumanPlayer** as well, which simply asks the user to input the moves and validates them.

3.2 Approach

My idea was to implement *MiniMax* with alpha-beta pruning, as it is the closest there is to an optimal strategy.

The problem with *MiniMax* for this kind of games is that it is very slow an inefficient during the first moves. The game trees are extremely large and it's very easy to exhaust the resources of a regular PC. For suppressing this issue, I thought about activating *MiniMax* at a later point of the game where the generated game trees wouldn't be as large.

An extension to this implementation is that the user can choose if, when reaching the depth limit of the Tree Search, the Player should consider the branch as a loss (thus, give a -1 reward) or continue the search using MCTS (Monte Carlo sampling) in order to have a chance of exploiting deeper game trees.

3.3 Design

- The turns in Quarto have a particular structure. A player's turn includes an action concerning himself (placing the given piece) and an action that concerns the opponent (selecting a piece to give to him/her). For this reason, the **run** game loop is not like **Nim**'s game loop, in which each iteration encapsulates the full action of a single player and switches player at the end of it. Instead, the player switching is done in the middle of the loop, so that a "full" turn starts in the second half of the iteration and ends in the first half of the following one.

  Therefore, I specify a **QuartoPly** to be a two-action ply:

1. Placing the given piece
2. Selecting piece for the opponent's first action

- As it was requested not to modify the Quarto library, my Minimax implementation accounts for the player switching externally (i.e. it doesn't alter the **current_player** internal variable of the Quarto instance).

- The **MiniMaxPlayer** computes the full *QuartoPly* inside the **place_piece()** method only (calling the *MiniMax* algorithm). Then, it stores the piece it will select on an internal variable (**piece_to_give**). The **choose_piece()** method, being the second action of its ply, only reads said internal variable when called (*MiniMax* is not called).

- **MiniMaxPlayer** can go first or second. If it goes first, the **choose_piece()** method will be called directly: It should read a value generated by a previous *MiniMax* call, but there hasn't been one yet. This issue is fixed by simply initializing the **piece_to_give** variable randomly (i.e. choosing a random piece). This is effective since in Quarto this first move does not have an impact on the chances of winning. Every piece is "worth" the same (same number of winning combinations can be made with all 16 pieces) and the board is empty at that moment.

- The **MiniMaxPlayer** contains a counter (**softTurns**) that initializes with a value given by the user and decreases each time **choose_piece()** is called. Before this counter hits 0, the moves are done either at random or using a fixed policy that will be defined in the next point. When it does hit 0, the endgame starts and the player calls *MiniMax* algorithm on each of its following turns.

- It is possible that an opponent (even playing randomly) gets to a very advantageous position or even wins against the **MiniMaxPlayer** on the first turns, before our player starts using *MiniMax*.

  There is definitely room for improvement on these initial moves. Here are some possible fixed policies for improving the early game performance of our player:

  – Instead of pure random, the moves can be made in a way that they try to "stall" the game and make it longer, using some fixed-rule policies. Here are some examples:

    * Placing the given piece in a position that shares row, column or diagonal the least with other pieces.
    * Selecting a piece that shares the least amount of common attributes with the pieces present already on the board.

3.4 Code

minimax.py

```python
import quarto
import random
from functools import cache
from typing import Callable
from collections import namedtuple
from itertools import product
import copy

QuartoPly = namedtuple('QuartoPly', ['placing_coords', 'selected_piece'])

def quartoPlying_new(game: quarto.Quarto, ply: QuartoPly) -> quarto.Quarto:
    game_copy = copy.deepcopy(game) # Avoid affecting the original status of the board

    game_copy.place(*ply.placing_coords)
    if ply.selected_piece != -1:
        # If there is a selected piece
        game_copy.select(ply.selected_piece)

    return game_copy
```

```python

def possible_moves(game: quarto.Quarto):
    '''A move consists on [placing the selected piece, selecting piece for opponent]'''
    selected_piece_idx = game.get_selected_piece()
    board = game.get_board_status()

    free_spaces = [(x, y) for x, y in product(range(4), range(4)) if board[y, x] == -1]
    if len(available_pieces := [p for p in range(16) if p not in board and p != selected_piece_idx]) == 0:
        available_pieces = [-1] # This is the case where the current player has been given the last piece
                                # of the whole board, so the piece he selects is '-1' instead of an empty list

    # Return a generator
    return (
        QuartoPly((x, y), p) for (x, y), p in product(free_spaces, available_pieces)
    )

@cache
def minimax(game: quarto.Quarto, maximizingPlayer: bool, maxDepth=None, alpha=-1, beta=1, MC=False):
    if MC:
        monteCarlo = False
    if maxDepth <= 0 or game.check_finished() or game.check_winner() != -1:
        if game.check_winner() != -1: # If there has been a winner
            return 1 if not maximizingPlayer else -1 # If it is not Max turn, it means Max won with his previous ply
        elif game.check_finished(): # If the game ended in a tie
            return 0

        # If maximum depth is reached, do Monte Carlo tree search or return -1 (assume branch is a loss)
        if MC:
            monteCarlo = True
        else:
            return -1

    scores = []
    moves = possible_moves(game)
    if MC:
        if monteCarlo:
            moves = [random.choice(list(moves))]
    for child_ply in moves:
        scores.append(
            score := minimax(quartoPlying_new(game, child_ply), not maximizingPlayer,
                maxDepth - 1 if maxDepth is not None else None, alpha, beta, MC)
        )
        if maximizingPlayer:
            alpha = max(alpha, score)
        else:
            beta = min(beta, score)
        if beta <= alpha: # In Nim, the case of beta < alpha won't happen (both are either +1 or -1)
            break

    return (max if maximizingPlayer else min)(scores)

def minimax_strategy(maxDepth=None, alpha=-1, beta=1, MC=False) -> Callable:

    def best_possible_move_minimax(game: quarto.Quarto) -> QuartoPly:
        return max(
            (minimax(quartoPlying_new(game, child_ply), False, maxDepth=maxDepth, alpha=alpha, beta=beta, MC=MC), child_ply)
            for child_ply in possible_moves(game)
        )[1] # Returns only the ply

```

```python
    return best_possible_move_minimax
```

players.py

```python
import logging
import random
import quarto
import minimax

class RandomPlayer(quarto.Player):
    """Random player"""

    def __init__(self, quarto: quarto.Quarto, *args) -> None:
        super().__init__(quarto)

    def choose_piece(self) -> int:
        return random.randint(0, 15)

    def place_piece(self) -> tuple[int, int]:
        return random.randint(0, 3), random.randint(0, 3)

class HumanPlayer(quarto.Player):
    ''' Human player, will ask the user to input the moves '''

    def __init__(self, quarto: quarto.Quarto, *args) -> None:
        super().__init__(quarto)

    def choose_piece(self) -> int:
        pieceIndex = None
        # Need to check the validity of the piece index here, since select() method doesn't do so.
        while True:
            logging.warning('Choose a piece for the opponent (in range [0, 15]): ')
            pieceIndex = input()
            try:
                pieceIndex = int(pieceIndex)
            except:
                logging.warning('Input is invalid!')
                continue
            if pieceIndex in range(16):
                break
            logging.warning('Please choose a valid piece index.')
        return pieceIndex

    def place_piece(self) -> tuple[int, int]:
        x = None
        y = None
        while True:
            logging.warning('Place the piece (format: x, y) (your position may be valid but already occupied): ')
            x_y = input()
            try:
                x, y = [int(n) for n in x_y.split(', ')]
            except:
                logging.warning('Input is invalid!')
                continue
            if x in range(4) and y in range(4):
                break
            logging.warning('Please choose a valid position.')
        return x, y

class MiniMaxPlayer(quarto.Player):
    '''
    MiniMax with Alpha-Beta pruning player with a twist: starts the first "randomTurns" turns randomly,
    "MC" flag determines if doing MCTS once maxDepth is reached, or if considering the branch as lost
    '''

```

```python
    def __init__(self, quarto: quarto.Quarto, maxDepth=None, alpha=-1, beta=1, softTurns=0, MC=False, slowStart=False) -> None:
        super().__init__(quarto)
        self.last_placement = (random.randint(0, 3), random.randint(0, 3))
        self.piece_to_give = random.randint(0, 15) # Selecting the first piece for the opponent randomly
        self.maxDepth = maxDepth
        self.alpha = alpha
        self.beta = beta
        self.softTurns = softTurns
        self.MC = MC
        self.slowStart = slowStart

    def choose_piece(self) -> int:
        if self.softTurns == 0:
            # Piece was selected before
            return self.piece_to_give
        else:
            # Second part of the turn
            # Need to ensure VALID turns! Otherwise, randomTurns counter will reach zero before
            # self.piece_to_give is properly initialized
            randomPiece = random.randint(0, 15)

            # Instead of random, compute a "score" array for all available pieces. The scores represents how
            # similar it is to the pieces already present on the board
            if self.slowStart:
                available_pieces = []
                pieces_on_board = []
                scores = []
                for n in range(16):
                    if n in self.get_game().get_board_status():
                        pieces_on_board.append(n)
                    else:
                        available_pieces.append(n)
                for avP in available_pieces:
                    bin_avP = self.get_game().get_piece_charachteristics(avP).binary
                    similarities = [0, 0, 0, 0]
                    for boardP in pieces_on_board:
                        bin_boardP = self.get_game().get_piece_charachteristics(boardP).binary
                        tmp_sim = [int(not a ^ b) for a, b in zip(bin_avP, bin_boardP)]
                        similarities = [a + b for a, b in zip(similarities, tmp_sim)]
                    scores.append((avP, sum(similarities)))
                self.softTurns -= 1
                return min(scores, key=lambda x: x[1])[0]

            if randomPiece not in self.get_game().get_board_status():
                self.softTurns -= 1
            if self.softTurns == 0:
                print('It\'s tryhard time.')
            return randomPiece

    def place_piece(self) -> tuple[int, int]:
        if self.softTurns == 0:
            # Place piece should update the player internal state, since it is the first action in the ply
            strategy = minimax.minimax_strategy(self.maxDepth, self.alpha, self.beta, MC=self.MC)
            (x, y), p = strategy(self.get_game())
            self.last_placement = (x, y)
            self.piece_to_give = p
            return x, y
        else:
            if self.slowStart:
                available_slots = []
                occupied_slots = []
```

```python
                scores = []
                for y in range(4):
                    for x in range(4):
                        if self.get_game().get_board_status()[y, x] == -1:
                            available_slots.append((x, y))
                        else:
                            occupied_slots.append((x, y))
                for avSlot in available_slots:
                    mainDiag = False
                    revDiag = False
                    sharing_score = 0
                    if avSlot[0] == avSlot[1]:
                        mainDiag = True
                    elif (avSlot[0] + avSlot[1]) == 3:
                        revDiag = True
                    for ocSlot in occupied_slots:
                        if ocSlot[0] == avSlot[0] or ocSlot[1] == avSlot[1]:
                            sharing_score += 1
                        if mainDiag:
                            if ocSlot[0] == ocSlot[1]:
                                sharing_score += 1
                        elif revDiag:
                            if (ocSlot[0] + ocSlot[1]) == 3:
                                sharing_score += 1
                    scores.append((avSlot, sharing_score))
                return min(scores, key=lambda x: x[1])[0]

            return random.randint(0, 3), random.randint(0, 3)
```

evaluate.py

```python
import quarto
import logging
from typing import Callable

def evaluate_player(player: Callable, args1, opponent: Callable, args2, NUM_MATCHES: int) -> float:
    '''
    Play multiple games against a given opponent, at each match the starting player changes.
    '''
    logging.info(f"Evaluating {player.__name__} against {opponent.__name__} over {NUM_MATCHES} matches ...")

    won = 0
    myPlayer_idx = 0

    for m in range(NUM_MATCHES):
        logging.debug(f'Match {m + 1}')
        game = quarto.Quarto() # Reset the game
        players = (player(game, *args1), opponent(game, *args2))
        game.set_players(players if myPlayer_idx == 0 else players[::-1]) # Invert the starting player
        winner = game.run()
        if winner == myPlayer_idx:
            won += 1
        myPlayer_idx = 1 - myPlayer_idx
    return won / NUM_MATCHES
```

main.py

```python
# Free for personal or classroom use; see 'LICENSE.md' for details.
# https://github.com/squillero/computational-intelligence

import logging
import argparse
import quarto
from collections import namedtuple
from players import *
from evaluate import *
```

```python

QuartoPly = namedtuple('QuartoPly', ['placing_coords', 'selected_piece'])

def main():
    # Run a single match
    game = quarto.Quarto()
    game.set_players((MiniMaxPlayer(game, maxDepth=3, softTurns=3, MC=False,
    slowStart=True), MiniMaxPlayer(game, maxDepth=3, softTurns=4, MC=False,
    slowStart=False)))
    winner = game.run()
    logging.warning(f"Winner: player {winner}")

    # Evaluate
    player = MiniMaxPlayer
    args1 = (3, -1, 1, 2, False, True)
    opponent = RandomPlayer
    args2 = (None,)
    N_MATCHES = 40
    winRate = evaluate_player(player, args1, opponent, args2, N_MATCHES)
    logging.info(f'win rate = {round(winRate * 100, 3)}%')


if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--verbose', action='count',
                        default=0, help='increase log verbosity')
    parser.add_argument('-d',
                        '--debug',
                        action='store_const',
                        dest='verbose',
                        const=2,
                        help='log debug messages (same as -vv)')
    args = parser.parse_args()

    if args.verbose == 0:
        logging.getLogger().setLevel(level=logging.WARNING)
    elif args.verbose == 1:
        logging.getLogger().setLevel(level=logging.INFO)
    elif args.verbose == 2:
        logging.getLogger().setLevel(level=logging.DEBUG)

    main()
```

3.5 Results

The best performing model (accounting for speed as well) against a **RandomPlayer** seems to be:

- Depth limit = 3
- Soft turns = 3
- MC = False

Ideally, setting a very high depth limit would give a better performance but the agent would be unacceptably slow. It is important to consider that the fixed-rule policy moves don't benefit our player, they just avoid the case of the player quickly falling into an early loss due to the randomicity of the moves. Thus, it is still a good idea to keep the **softTurns** counter as small as possible.

MCTS does not really improve performance against a **RandomPlayer** since this opponent usually doesn't reach "long" games against our player, so that considering deeper nodes in the Search Tree is not very relevant.

When making the **MiniMaxPlayer** play against a fellow **MiniMaxPlayer**, I noticed that the most significant factor for winning consistently becomes the number of soft turns. The player that activates *MiniMax* faster, wins.

3.6 Further improvements (not implemented)

- The first moves could be made by exploiting Reinforcement Learning: The Q-table could be trained beforehand against a Random opponent for several episodes and then used for the first N moves, then going back to *MiniMax* on the endgame.

4 Conclusion