COMPUTATIONAL INTELLIGENCE

Final Log - 2023/2024

Written by Angelo Iannielli - s317887

LAB 1

Set Covering

Introduction

The aim of the first lab was to solve the "Set Covering Problem" using the ***A* algorithm***. The set covering is a combinatorial optimization problem where the goal is to select a set of subsets from a larger set in such a way that every element of the larger set is covered by at least one of the selected subsets.

In particular, we were tasked with implementing a new version of the ***H function***, which is a heuristic function estimating the minimum cost to reach the solution from a current state to the goal state. The function g is a function measuring the actual cost to reach the current state and was assumed to be the number of sets taken.

Code

```python
from random import random
from functools import reduce
from collections import namedtuple
from queue import PriorityQueue
from math import ceil
import numpy as np
```

```python
PROBLEM_SIZE = 15
NUM_SETS = 25
SETS = tuple(
    np.array([random() < 0.3 for _ in range(PROBLEM_SIZE)])
    for _ in range(NUM_SETS)
)
State = namedtuple('State', ['taken', 'not_taken'])
```

```python
def goal_check(state):
    return np.all(reduce(
        np.logical_or,
        [SETS[i] for i in state.taken],
        np.array([False for _ in range(PROBLEM_SIZE)]),
    ))

def covered(state):
    return reduce(
        np.logical_or,
        [SETS[i] for i in state.taken],
        np.array([False for _ in range(PROBLEM_SIZE)]),
    )

def g(state):
    return len(state.taken)



# number of positions to cover to reach the goal
def h1(state):
    return PROBLEM_SIZE - sum(
        covered(state))
```

```python
def h2(state):
    covered_tiles = sum(covered(state))
    if covered_tiles == PROBLEM_SIZE:
        return 0
    return 1 / covered_tiles if covered_tiles != 0 else 1
```

```python
# We only considered the sets not taken,
# so as not to be influenced by the existence of large sets which have already been taken
def h3(state):
    not_taken_sets = [s for i, s in enumerate(SETS) if i not in state.taken]
    largest_set_size = max(sum(s) for s in not_taken_sets) # select the larget tiles (more 
number of true)
    missing_size = PROBLEM_SIZE - sum(covered(state)) # evaluates the number of tiles that are 
not covered
    optimistic_estimate = ceil(missing_size / largest_set_size) # estimate the number of set 
that are missing for the solution in a optimistic way
    # if the largest set is 5 and the missing size is 10 --> "maybe" 2 sets are missing 
(optimistic assumption)
    return optimistic_estimate
```

```python
def f1(state):
    cost_1 = g(state)
    cost_2 = h1(state)

    return cost_1 + cost_2

# since h2 is a value between 0 and 1, we multiply it by 0.1 to make it more significant
def f2(state):
    cost_1 = 0.1*g(state)
    cost_2 = h2(state)

    return cost_1 + cost_2

def f3(state):
    cost_1 = g(state)
    cost_2 = h3(state)

    return cost_1 + cost_2

assert goal_check(
    State(set(range(NUM_SETS)), set())
), "Problem not solvable"
```

Solution with h1

```
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f1(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((f1(new_state), new_state))
    _, current_state = frontier.get()

print(
    f"Solved in {counter:,} steps ({len(current_state.taken)} tiles)"
)
```

Solution with h2

```
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f2(state), state))

counter = 0
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((f2(new_state), new_state))
    _, current_state = frontier.get()

print(
    f"Solved in {counter:,} steps ({len(current_state.taken)} tiles)"
)
```

Solution with h3

```
frontier = PriorityQueue()
state = State(set(), set(range(NUM_SETS)))
frontier.put((f3(state), state))

counter = 0
```

```
_, current_state = frontier.get()
while not goal_check(current_state):
    counter += 1
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        frontier.put((f3(new_state), new_state))
    _, current_state = frontier.get()

print(
    f"Solved in {counter:,} steps ({len(current_state.taken)} tiles)"
)
```

The code is written by me and Nicolò Caradonna (s316993).

LAB 1 1/2

Halloween Challenge

Introduction

The objective of the "second" lab was to solve a set covering problem in the best possible manner. We were provided with a function that generated a set of problem instances, and the choice of algorithms to use was left to our discretion. The problem instances were configured through a series of parameters provided in the prompt.

In particular, the parameters was:

num_points *= [100, 1_000, 5_000]*
num_sets *= num_points*
density *= [.3, .7]*

Code

```
from itertools import product
import numpy as np
from scipy import sparse
from random import random, choice, randint, seed
from functools import reduce
from copy import copy
import math
import matplotlib.pyplot as plt
```

```
num_points = [100, 1_000, 5_000]
num_sets = num_points
density = [0.3, 0.7]
```

```
points = num_points[0]
sets = num_sets[0]
den = density[0]
iterations = 100
def make_set_covering_problem(num_points, num_sets, density):
    """Returns a sparse array where rows are sets and columns are the covered items"""
    seed(num_points * 2654435761 + num_sets + density)
    sets = sparse.lil_array((num_sets, num_points), dtype=bool)
    for s, p in product(range(num_sets), range(num_points)):
        if random() < density:
            sets[s, p] = True
    for p in range(num_points):
        sets[randint(0, num_sets - 1), p] = True
    return sets
```

```
SETS = make_set_covering_problem(points, sets, den)
```

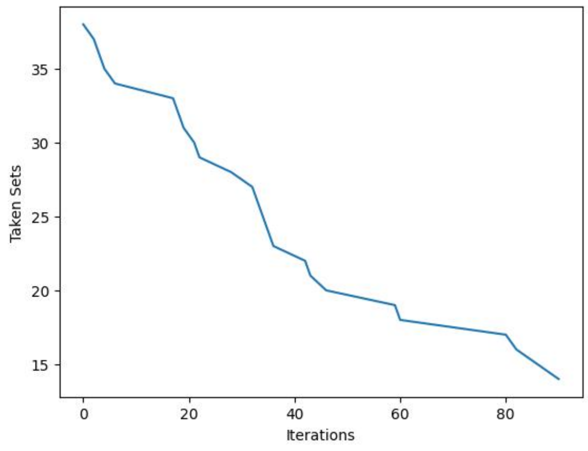
Solution (Hill-Climbing)

```
# Taken from Giovanni Squillero's notebook on Github
def evaluate(state):
    cost = sum(state)
    valid = np.all(
        reduce(
            np.logical_or,
            [SETS.getrow(i).toarray().flatten() for i, t in enumerate(state) if t],
            np.array([False for _ in range(points)]),
        )
    )
    return valid, -cost if valid else 0
```

```
def tweak(state):
    new_state = copy(state)
    index = randint(0, sets - 1)
    new_state[index] = not new_state[index]

    return new_state
```

```
current_state = [choice([True, False]) for _ in range(sets)]
# current_state = [choice([False]) for _ in range(num_sets[1])]
taken_sets = []

iteration_sets = []
for step in range(iterations):
    new_state = tweak(current_state)
    if evaluate(new_state) >= evaluate(current_state):
        current_state = new_state
        # print(current_state, evaluate(current_state))
        taken_sets.append(-evaluate(current_state)[1])
        iteration_sets.append(step)
        print("Step: " + str(step) + " Current state: " + str(evaluate(current_state)))
print("Final state:", evaluate(current_state))
```

```
plt.plot(iteration_sets, taken_sets)
plt.xlabel("Iterations")
plt.ylabel("Taken Sets")
plt.show()
```

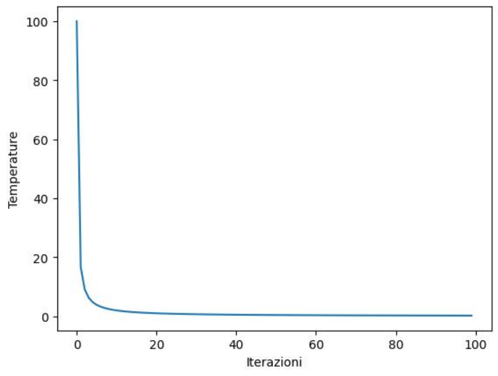


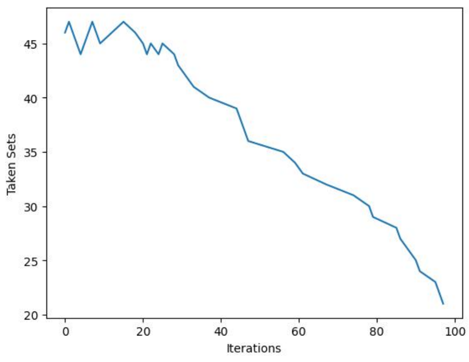
Solution (Simulated Annealing)

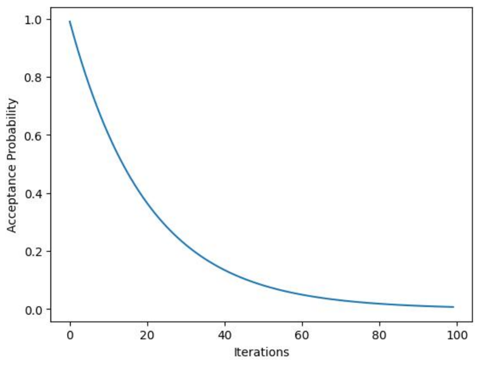
```
def acceptance_probability(current_solution, tweaked_solution, temp):
    x = -abs(current_solution[1] - tweaked_solution[1]) / temp
    return math.exp(x)
```

```
current_state = [choice([True, False]) for _ in range(sets)]
# current_state = [choice([False]) for _ in range(num_sets[1])]

temp_array = []
probability_array = []
taken_sets = []
iteration_sets = []

for step in range(iterations):
    new_state = tweak(current_state)
    temp = iterations / (5 * step + 1)
    temp_array.append(temp)
    p = acceptance_probability(evaluate(current_state), evaluate(new_state), temp)
    probability_array.append(p)

    if evaluate(new_state) >= evaluate(current_state) or random() < p:
        current_state = new_state
        # print(current_state, evaluate(current_state))
        taken_sets.append(-evaluate(current_state)[1])
        iteration_sets.append(step)
        print("Step: " + str(step) + " Current state: " + str(evaluate(current_state)))

print("Final state:", evaluate(current_state))
```

```
plt.plot(iteration_sets, taken_sets)
plt.xlabel("Iterations")
plt.ylabel("Taken Sets")
plt.show()

plt.plot(range(iterations), temp_array)
plt.xlabel("Iterazioni")
plt.ylabel("Temperature")
plt.show()

plt.plot(range(iterations), probability_array)
plt.xlabel("Iterations")
plt.ylabel("Acceptance Probability")
plt.show()
```







Solution (Tabu Search)

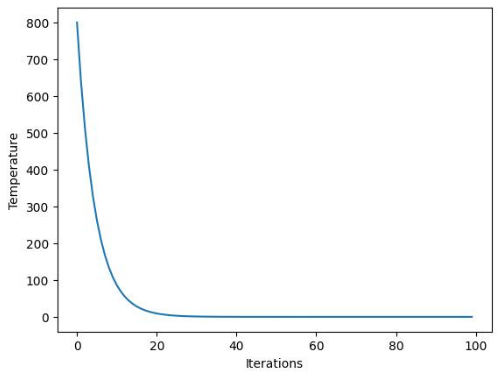
```
temperature = 1000
cooling_rate = 0.8
taboo_list = []
temp_array = []
iteration_sets = []
probability_array = []
```

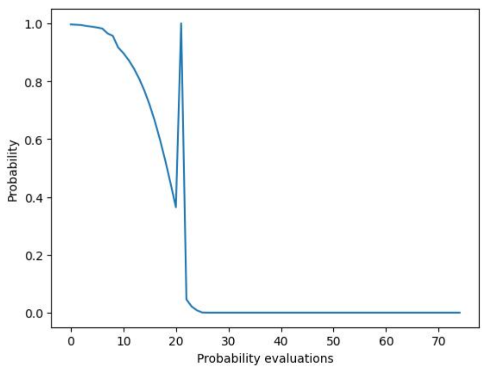
```
def find_greatest_set(x):
    return x.sum(axis=1).argmax()


def evaluate_2(state):
    cost = sum(state)

    elem_covered = reduce(
        np.logical_or,
        [SETS.getrow(i).toarray() for i, t in enumerate(state) if t],
        np.array([False for _ in range(points)]),
    )

    valid = np.all(elem_covered)

    num_elem_covered = np.count_nonzero(elem_covered)

    return valid, num_elem_covered, -cost

def tweak_2(state):
    new_state = copy(state)

    while new_state in taboo_list:
        index = randint(0, sets - 1)
        new_state[index] = not new_state[index]

    taboo_list.append(new_state)
    return new_state
```

```
## Initialize the taboo list
taboo_list.clear()

## Find the set that cover the most num of elements and use it as starting point
current_solution = [False] * sets
current_solution[find_greatest_set(SETS)] = True
current_cost = evaluate_2(current_solution)

# Memorize that as the best solution for the moment
best_solution = [True] * sets
best_cost = (True, points, -sets)

# Insert the starting point into taboo list
taboo_list.append(current_solution)
```

```
for step in range(iterations):
    # Find a new possible solution
    new_state = tweak_2(current_solution)
    # print(new_state)
    # Evaluate the cost
    new_cost = evaluate_2(new_state)
    print(new_cost)

    # Calculate deltaE using the number of taken elements
    deltaE = - ( new_cost[1] - current_cost[1] )
    print(deltaE)

    if deltaE == 0:
        # Calculate deltaE using the number of taken sets
        deltaE = - ( new_cost[2] - current_cost[2] )

    # The solution is better
    if deltaE < 0:
        current_solution = new_state
        current_cost = new_cost

        if current_cost[2] > best_cost[2] and current_cost[0] == True:
            best_solution = current_solution
            best_cost = current_cost
    else:
        probability = math.exp(-deltaE / temperature)
        probability_array.append(probability)

        if random() < probability:
                current_solution = new_state
                current_cost = new_cost

    temperature *= cooling_rate
    temp_array.append(temperature)
    iteration_sets.append(step)
```

```
plt.plot(range(iterations), temp_array)
plt.xlabel("Iterations")
plt.ylabel("Temperature")
plt.show()

plt.plot(range(len(probability_array)), probability_array)
plt.xlabel("Probability evaluations")
plt.ylabel("Probability")
plt.show()
```

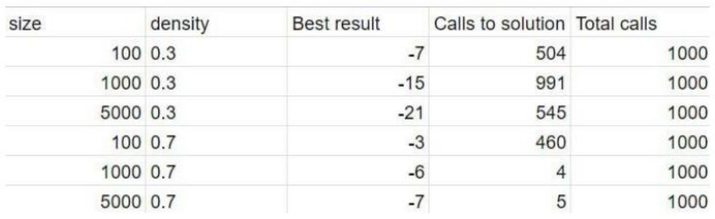




The code is written by me and Nicolò Caradonna (s316993).

Results (Tabu Search)

Here are the results of the solution obtained using tabu search, which proved to be the best among the implemented alternatives.



size	density	Best result	Calls to solution	Total calls
100	0.3	-7	504	1000
1000	0.3	-15	991	1000
5000	0.3	-21	545	1000
100	0.7	-3	460	1000
1000	0.7	-6	4	1000
5000	0.7	-7	5	1000

LAB 2

Halloween Challenge

Introduction

The second lab required the implementation of an algorithm capable of playing Nim. Nim is a strategy game played between two players taking turns. It is played with a set of piles of objects. At the beginning, there are several piles of objects, and players alternate turns, each taking any number of objects from a single pile. The rules imposed by the prompt are that the last player to take an object loses.

To implement this, we utilized a model based on an Evolutionary Algorithm.

Code

```python
import logging
from pprint import pprint, pformat
from typing import Callable
from collections import namedtuple
import random
from copy import deepcopy
import matplotlib.pyplot as plt
import random
import numpy as np
```

(Game Class)

```python
NUM_ROWS = 5
K = None
NUM_MATCHES = 200
λ = 20
σ = 0.1
GENERATION_SIZE = 500 // λ
random.seed(42)
```

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

```python
class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        # Initialize the Nim object with given number of rows and an optional maximum object
limit
        self._rows = [
            i * 2 + 1 for i in range(num_rows)
        ]  # Create a list of odd numbers as row sizes
        self._k = k  # Store the maximum object limit

    def __bool__(self):
        # Return True if there are objects remaining in the game, False otherwise
        return sum(self._rows) > 0

    def __str__(self):
        # Return a string representation of the object
        return "<" + " ".join(str(_) for _ in self._rows) + ">"

    @property
    def rows(self) -> tuple:
        # Return the rows as a tuple
        return tuple(self._rows)

    def nimming(self, ply: Nimply) -> None:
        # Perform a nimming move by removing objects from a specified row
        row, num_objects = ply  # Unpack the tuple
        assert (
            self._rows[row] >= num_objects
        )  # Check if the specified row has enough objects
        assert (
            self._k is None or num_objects <= self._k
        )  # Check if the number of objects is within the maximum limit
        self._rows[
            row
        ] -= num_objects  # Subtract the number of objects from the specified row
```

(Sample and silly strategies)

```python
def pure_random(state: Nim) -> Nimply:
    """A completely random move"""
    # Select a row that has at least one object remaining
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    # Randomly choose a number of objects to remove from the selected row
    num_objects = random.randint(1, state.rows[row])
    # Create and return a Nimply object representing the chosen move
    return Nimply(row, num_objects)
```

```python
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    # Generate a list of possible moves
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)]
    # Select the move with the maximum number of objects from the lowest row
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))
```

```python
def nim_sum(state: Nim) -> int:
    tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in state.rows])
    xor = tmp.sum(axis=0) % 2
    return int("".join(str(_) for _ in xor), base=2)


def analize(raw: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = dict()
    for ply in (Nimply(r, o) for r, c in enumerate(raw.rows) for o in range(1, c + 1)):
        tmp = deepcopy(raw)
        tmp.nimming(ply)
        cooked["possible_moves"][ply] = nim_sum(tmp)
    return cooked
```

```python
def optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].items() if ns != 0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply
```

```python
def state_info(state: Nim) -> dict:
    info = dict()
    info["possible_moves"] = [
        (r, o) for r, c in enumerate(state.rows) for o in range(1, c + 1)
    ]
    info["shortest_row"] = min(
        (x for x in enumerate(state.rows) if x[1] > 0), key=lambda y: y[1]
    )[0]
    info["longest_row"] = max((x for x in enumerate(state.rows)), key=lambda y: y[1])[0]
    info["random_row"] = random.choice([r for r, c in enumerate(state.rows) if c > 0])


    return info
```

```python
def evolved_strategy(genome) -> Callable:
    strategy_dict = {0: "shortest", 1: "longest", 2: "random", 3: "half_random", 4:
"one_random"}

    def adaptive(state: Nim) -> Nimply:
        data = state_info(state)

        # select a strategy in a random way wheighed by the genome
        selected_strategy = random.choices(range(len(genome)), weights=genome)[0]
        selected_strategy = strategy_dict[selected_strategy]
        if selected_strategy == "shortest":
            ply = Nimply(
                data["shortest_row"],
                random.randint(1, state.rows[data["shortest_row"]]),
            )
        elif selected_strategy == "longest":
            ply = Nimply(
                data["longest_row"], random.randint(1, state.rows[data["longest_row"]])
            )
        elif selected_strategy == "random":
            ply = Nimply(
                data["random_row"], random.randint(1, state.rows[data["random_row"]])
            )
        elif selected_strategy == "half_random":
            ply = Nimply(data["random_row"], (state.rows[data["random_row"]] // 2 + 1))

        elif selected_strategy == "one_random":
            ply = Nimply(data["random_row"], 1)
        # else:
        #     ply = optimal(state)
        return ply

    return adaptive
```

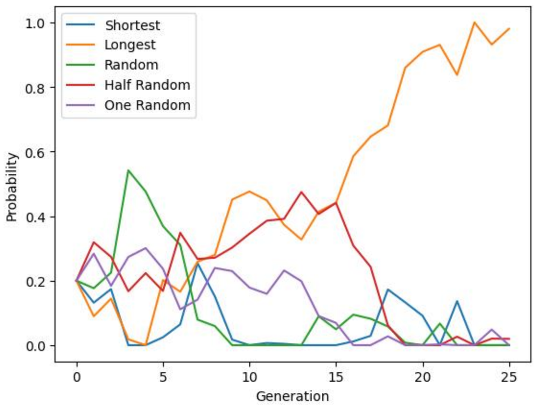
```
# In the fitness function we play Nim for NUM_MATCHES times where the player is:
# adaptive: for each move, choose a rule in a random way wheighed by the genome
# optimal: choose the optimal move


# Since "optimal" strategy is our upper bound, we can find the best individual among
population by comparing it with an individual that plays always with the optimal solution

def fitness(adaptive: Callable) -> int:
    won = 0
    opponent = (adaptive, optimal)

    for _ in range(NUM_MATCHES):
        nim = Nim(NUM_ROWS)
        player = 0
        while nim:
            ply = opponent[player](nim)
            nim.nimming(ply)  # perform the move
            player ^= 1

        if player == 0:
            won += 1

    return won  # return the number of matches won
```

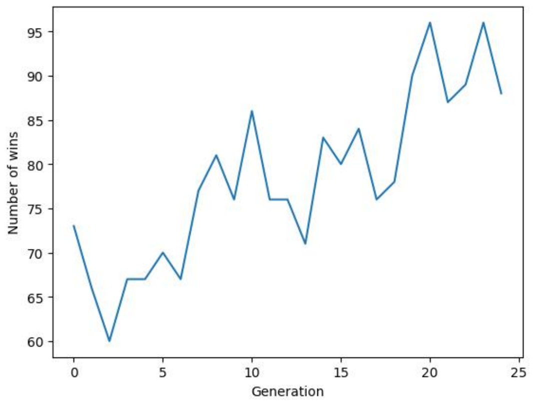
```
def generate_offsprings(offspring) -> list:
    output = []

    for _ in range(λ):
        new_offspring = [
            np.clip(val + np.random.normal(0, σ), 0, 1) for val in offspring
        ]

        current_sum = sum(new_offspring)

# Normalize the sum to 1 if it is not already
        if current_sum != 1:
            scale_factor = 1 / current_sum
            # Apply scale factor to each value
            values = [val * scale_factor for val in new_offspring]
        else:
            values = new_offspring

        output.append(values)

    return output
```

Solution with (1,λ)-ES

```
current_solution = (0.20, 0.20, 0.20, 0.20, 0.20)


choosen_probability = list()
solutions_list = list()
for n in range(GENERATION_SIZE):
    # offspring <- select λ random points mutating the current solution
    # print("Starting probability for generation", n+1, "is:", current_solution)
    offsprings = generate_offsprings(current_solution)
    offsprings.append(current_solution)
    # evaluate and select best

    evals = [
        (offspring, fitness(evolved_strategy(offspring))) for offspring in offsprings
    ]

    evals.sort(key=lambda x: x[1], reverse=True)
    # pprint(evals)

    current_solution = evals[0][0]
    choosen_probability.append(current_solution)
    solutions_list.append(evals[0][1])

    print(f"Best result for generation {n+1} is:", evals[0])

val = np.array([[0.20], [0.20], [0.20], [0.20], [0.20]])
curve_names = ["Shortest", "Longest", "Random", "Half Random", "One Random"]
choosen_probability = np.array(choosen_probability)

for i in range(GENERATION_SIZE):
    val = np.hstack((val, choosen_probability[i].reshape(-1, 1)))

for i in range(5):
    plt.plot(range(GENERATION_SIZE + 1), val[i], label=curve_names[i])

plt.xlabel("Generation")
plt.ylabel("Probability")
plt.legend()
plt.show()

plt.plot(range(GENERATION_SIZE), solutions_list)
plt.xlabel("Generation")
plt.ylabel("Number of wins")
```

```python
plt.show()
```

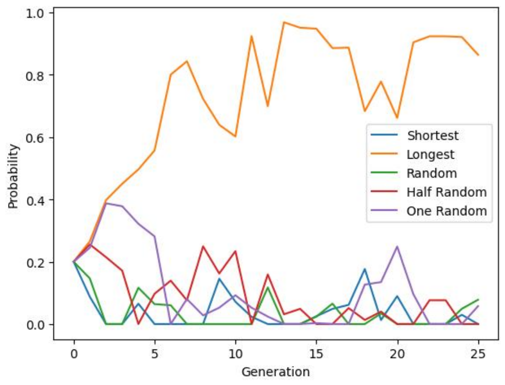


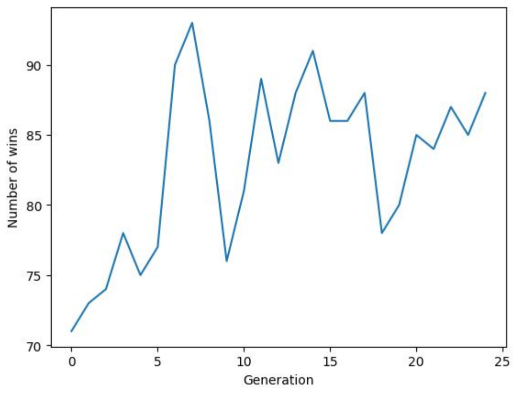


Solution with Adaptive (1,λ)-ES

```python
current_solution = (0.20, 0.20, 0.20, 0.20, 0.20)
choosen_probability = list()
solutions_list = list()
stats = [0, 0]
counter = 0
for n in range(GENERATION_SIZE):
    print("Sigma for generation", n + 1, "is:", σ)
    offsprings = generate_offsprings(current_solution)
    offsprings.append(current_solution)

    evals = [
        (offspring, fitness(evolved_strategy(offspring))) for offspring in offsprings
    ]
    previous_solution = evals[λ]
    for i in range(λ):
        if evals[i][1] > previous_solution[1]:
            counter += 1

    stats[1] += counter
    stats[0] += λ

    if (n + 1) % 5 == 0:
        if stats[1] / stats[0] < 1 / 5:
            σ /= 1.1
        elif stats[1] / stats[0] > 1 / 5:
            σ *= 1.1


    evals.sort(key=lambda x: x[1], reverse=True)

    # pprint(evals)

    current_solution = evals[0][0]
    choosen_probability.append(current_solution)
    solutions_list.append(evals[0][1])

    print(f"Best result for generation {n+1} is:", evals[0])

val = np.array([[0.20], [0.20], [0.20], [0.20], [0.20]])
curve_names = ["Shortest", "Longest", "Random", "Half Random", "One Random"]
choosen_probability = np.array(choosen_probability)
```

```
for i in range(GENERATION_SIZE):
    val = np.hstack((val, choosen_probability[i].reshape(-1, 1)))

for i in range(5):
    plt.plot(range(GENERATION_SIZE + 1), val[i], label=curve_names[i])

plt.xlabel("Generation")
plt.ylabel("Probability")
plt.legend()
plt.show()

plt.plot(range(GENERATION_SIZE), solutions_list)
plt.xlabel("Generation")
plt.ylabel("Number of wins")
plt.show()
```





The code is written by me and Nicolò Caradonna (s316993).

LAB 9

Fitness calls optimization

Introduction

The prompt for the third (*3) lab requires solving Problem instances 1, 2, 5, and 10 on 1000-loci genomes, using the fewest fitness calls possible.

The problem we need to solve is to optimize the fitness of our solution, minimizing the number of fitness calls. We should consider it as a black box so the extra information that the best solution is composed by all 1 will be ignored.

My Idea

I tried to implement an algorithm that use the concept of "islands" to group the solutions with similar values of fitness. To do this I implemented this things:

- An ***archipelago***, that is a list of island (3 in my algorithm)
- The ***islands***, that have some parameters to define their own population capacity, their mutation rate, the number of parents needed to have a new individual, the size of tournament (for the parent selection) and the interval of allowed fitness.
- The ***individuals***, that have 2 different ways to be created: the first one is random and it is used only to generate the first population, the second consists in a xover between the parents and a mutation (according to the island mutation rate).

Code

```python
from random import choices
from random import random, randint, sample
from collections import namedtuple
from copy import deepcopy, copy
from typing import List
import matplotlib.pyplot as plt

import lab9_lib
```

```python
# GLOBAL VARIABLES INITIALIZATION

LOCI = 1000
population = []

THRESHOLDS = 3
DIM_FIRST_POP = 100

GENERATIONS = 10000
gen = 1

problem = lab9_lib.make_problem(1)
```

```python
class Individual:
    def __init__(self):
        self.genome = []
        self.fitness = 0

    def generate_randomly(self):
        self.genome = choices([0, 1], k=LOCI)
        self.fitness = problem(self.genome)

    def generate_by_recombination(self, mut_rate, parents):

        # Define the dimension of the slices in the xover
        dim_slice = randint(1, LOCI // len(parents))

        for slice in range(LOCI // dim_slice):
            selected_parent = choices(parents, weights=[p.fitness for p in parents], k=1)
            genes = selected_parent[0].genome[slice * dim_slice : slice * dim_slice + 
dim_slice]
            self.genome.extend(genes)
        # Fill the genes out of the last slice
        selected_parent = choices(parents, weights=[p.fitness for p in parents], k=1)
        genes = selected_parent[0].genome[LOCI // dim_slice * dim_slice : LOCI]
        self.genome.extend(genes)

        for index in range(len(self.genome)):
            if random() < mut_rate :
                self.genome[index] = 1 if self.genome[index] == 0 else 0

    def estimate_fitness(self):
        self.fitness = problem(self.genome)

    def __str__(self):
        return f"{self.fitness:.2%} : {''.join(str(g) for g in self.genome)}"
```

Solution

```
class Island:
    def __init__(self, min_fitness, max_fitness, population, num_parents, mut_rate,
dim_population, dim_tournament, level):
        self.min_fitness = min_fitness
        self.max_fitness = max_fitness
        self.population = population
        self.num_parents = num_parents
        self.mut_rate = mut_rate
        self.dim_population = dim_population
        self.dim_tournament = dim_tournament
        self.level = level

    def parents_selection(self):
        # sorted_population = sorted(self.population, key=lambda ind: ind.fitness,
reverse=True)
        # return sorted_population[:self.dim_population]

        population_copy = copy(self.population)
        parents = []

        for _ in range(len(self.population)//self.dim_tournament):
            tournament = sample(population_copy, k= self.dim_tournament)
            tournament = sorted(tournament, key=lambda ind: ind.fitness, reverse=True)

            parents.append(tournament.pop(0))
            for loser in tournament:
                population_copy.remove(loser)

        return parents

    def survival_selection(self):
        sorted_population = sorted(self.population, key=lambda ind: ind.fitness, reverse=True)

        travellers = namedtuple('travellers', ['weak', 'strong'])
        weak_population = []
        strong_population = []
        new_sorted_population = []


        for ind in sorted_population:

            # If the individual has a low fitness it is demote to a lower island
            if ind.fitness < self.min_fitness:
                weak_population.append(ind)

            # If the individual has a high fitness it is promote to an higher island
            elif ind.fitness > self.max_fitness:
                strong_population.append(ind)

            else:
                new_sorted_population.append(ind)

        return travellers(weak_population, strong_population)

    def shrink_population(self):
        sorted_population = sorted(self.population, key=lambda ind: ind.fitness, reverse=True)
        self.population = sorted_population[:self.dim_population]

    def get_top_solution(self) -> Individual :
        if len(self.population) > 0:
            return self.population[0]
        else:
            return None

    def __str__(self):
        return f"Island {self.min_fitness:.2%}-{self.max_fitness:.2%} > POPULATION SIZE:
{len(self.population)}"
```

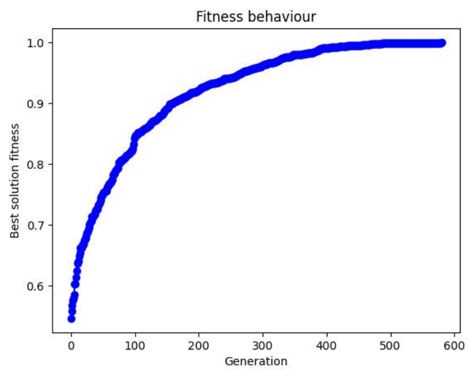
```
archipelago: List[Island] = []
solution = Individual()
```

```
archipelago: List[Island] = []
THRESHOLDS = 3
DIM_FIRST_POP = 80

solution = Individual()
best_solutions_fitness = []

for level in range(THRESHOLDS):
    archipelago.append(Island(min_fitness= 1 / THRESHOLDS * level, max_fitness= 1 / THRESHOLDS
* ( level + 1), population= [], num_parents= THRESHOLDS - level + 1, mut_rate= 1 / (10 **
(level + 1)), dim_population= DIM_FIRST_POP // ( level + 1 ), dim_tournament= THRESHOLDS + 2,
level= level))

# FIRST GENERATION
for _ in range(DIM_FIRST_POP):
    ind = Individual()
    ind.generate_randomly()
    archipelago[0].population.append(ind)

weak, strong = archipelago[0].survival_selection()

print("Weak dim: ", len(weak), " Strong dim: ", len(strong))

# Put the weak individuals in the correct island
for ind in weak:
    for island in archipelago:
        if ind.fitness > island.min_fitness and ind.fitness < island.max_fitness:
            island.population.append(ind)

# Bring the strong individual high
for ind in strong:
    for island in archipelago:
        if ind.fitness > island.min_fitness and ind.fitness < island.max_fitness:
            island.population.append(ind)

for island in archipelago:
    # Shrink the population to dim_population elements
    island.shrink_population()
    print(island)
    for ind in island.population:
        print(ind)

# Save the top solution
for island in reversed(archipelago):
    current_top_solution = island.get_top_solution()
    if current_top_solution:
        print("Top solution:", current_top_solution)
        solution = current_top_solution
        break

best_solutions_fitness.append(solution.fitness)
print(problem.calls)
```

```
GENERATIONS = 10000
gen = 1

# Iterate until a perfect solution is found or the number of generations is reached
while solution.fitness < 1.0 and gen < GENERATIONS:

    print("GENERATION", gen)

    # For each island, select a group of parents and generate the offsprings
    for island in archipelago:
        selected_parents = island.parents_selection()

        # The number of new offsprings is enough to fill the expected space of the island
        for _ in range(island.dim_population):

            # Every island needs a specific number of parents to create a new individual
            if len(selected_parents) > island.num_parents:
                parents = sample(selected_parents, k= island.num_parents)

                child = Individual()

                child.generate_by_recombination(island.mut_rate, parents)

                child.estimate_fitness()

                island.population.append(child)

    # For each island, there is a survival selection. The final number of individuals must be
<= dim_population
    for island in archipelago:

        # Every selection updates the population inside the island and returns two lists of
individuals that are promoted/demoted
        weak, strong = island.survival_selection()

        print("Island LVL", island.level, ">", "promoted:", len(strong), "demoted:", len(weak))

        for ind in weak:
            for index in range(0, island.level):
                island = archipelago[index]
                if ind.fitness > island.min_fitness and ind.fitness < island.max_fitness:
                    island.population.append(ind)

        for ind in strong:
            for index in range(island.level + 1, len(archipelago)):
                island = archipelago[index]
                if ind.fitness > island.min_fitness and ind.fitness < island.max_fitness:
                    island.population.append(ind)

    for island in archipelago:
            # Shrink the population to dim_population elements
            island.shrink_population()

            print(island)
```

```
    # Check the top solution
    current_top_solution = None

    # Start by searching from the highest level
    for island in reversed(archipelago):
        current_top_solution = island.get_top_solution()

        # When a solution is found, it must be the better (because it is in the first position
of the top level)
        if current_top_solution:
            print("Current top sol:", current_top_solution)

            # Check if the solution is better than the latest one
            if current_top_solution.fitness > solution.fitness :
                gain = (current_top_solution.fitness - solution.fitness)
                print(f"Found a better solution (+{gain:.2%}): {current_top_solution}")
                solution = current_top_solution
            break

    best_solutions_fitness.append(solution.fitness)
    print("Fitness called", problem.calls, "times")
    print("")

    gen += 1
```

```
# Crea il grafico
plt.plot(list(range(gen)), best_solutions_fitness, marker='o', linestyle='-', color='b')

# Aggiungi etichette e titoli
plt.xlabel('Generation')
plt.ylabel('Best solution fitness')
plt.title('Fitness behaviour')

# Visualizza il grafico
plt.show()
```



LAB 10

Tic-tac-toe

Introduction

Lab 10 asks to implement a reinforcement learning algorithm capable of playing tic tac toe.

To solve the problem, **magic_square** [2, 7, 6, 9, 5, 1, 4, 3, 8] was used: this made it possible to simplify the way in which victory (sum = 15) can be verified and also the decision to be made during one's turn. To implement an RL algorithm, it is necessary to provide rewards to be assigned to the agent: in my case I assigned **1** pt for **victory**, **-1** pt for **defeat**, and **0** pt for a **tie**.

Code

```
'''Import modules'''
from collections import namedtuple, defaultdict
from random import choice, shuffle
from copy import deepcopy
from typing import Type
from itertools import combinations
```

```
'''Define utilities'''
State = namedtuple('State', ['X', 'O'])
MAGIC_SQUARE = [2, 7, 6, 9, 5, 1, 4, 3, 8]
```

```
'''Player class'''
class Player:
    def __init__(self, name = "RandomPlayer"):
        self.name = name
        self.strategy = self.random_choice

    def random_choice(self, available_moves, our_state=None, opponent_state=None, 
policies=None):
        move = choice(available_moves)
        return move
    def trained_player(self, available_moves, our_state, opponent_state, policies):
        max_value = float('-inf')
        best_move = choice(available_moves)

        for move in available_moves:

            possible_state = our_state.copy()
            possible_state.append(move)
            rating = policies.get(frozenset(possible_state), frozenset(opponent_state))
            if rating > max_value:
                    max_value = rating
                    best_move = move

        return best_move

    def set_trained_player(self):
         self.strategy = self.trained_player
```

```

    def __str__(self) -> str:
        return f"{self.name}"
```

```
'''Game class'''
class TicTacToe:
    def __init__(self):
        self.board = MAGIC_SQUARE
        self.available_moves = MAGIC_SQUARE.copy() # Define the board
        self.X_boxes = []
        self.O_boxes = []

    def print_board(self):
        for row in range(3):

            if row != 0 and row != 3 :
                print("-------------")

            for col in range(3):
                i = row * 3 + col
                char = " "
                if self.board[i] in self.X_boxes:
                    char = "X"
                elif self.board[i] in self.O_boxes:
                    char = "O"

                print(f"| {char}", end=" ")


            print("|")

    def win_condition(self, moves_set):
        win_condition = any(sum(c) == 15 for c in combinations(moves_set, 3))
        return win_condition

    def play_game(self, playerA, playerB):

        players = [playerA, playerB]
        shuffle(players)

        playerX, playerO = players
        current_player = playerX
        print(f"{playerX} is X, {playerO} is O")

        trajectory = list()

        while self.available_moves:

            # Select a move using the player strategy
            move = current_player.strategy(self.available_moves)

            # Remove the move from the available
            self.available_moves.remove(move)

            if current_player == playerX :
                if self.win_condition(self.X_boxes):
                    self.X_boxes.append(move)
                    trajectory.append(State(self.X_boxes.copy(), self.O_boxes.copy()))
                    break
                else:
                    self.X_boxes.append(move)
                    trajectory.append(State(self.X_boxes.copy(), self.O_boxes.copy()))
                    current_player = playerO
            else:
                if self.win_condition(self.O_boxes):
                    self.O_boxes.append(move)
                    trajectory.append(State(self.X_boxes.copy(), self.O_boxes.copy()))
                    break
                else :
                    self.O_boxes.append(move)
                    trajectory.append(State(self.X_boxes.copy(), self.O_boxes.copy()))
                    current_player = playerX

        return trajectory

    def play_game_trained(self, policies, opponent):
        playerT = Player("TrainedPlayer")
        playerT.set_trained_player()

        players = [playerT, opponent]
        shuffle(players)

        current_player = players[0]
        winner = None

        while self.available_moves:

            # Select a move using the player strategy
            if current_player == playerT:
                move = current_player.strategy(self.available_moves, self.X_boxes,
self.O_boxes, policies) # Trained plays always X
                self.X_boxes.append(move)
                if self.win_condition(self.X_boxes):
                    winner = playerT
                    print("TRAINED PLAYER WON")
                    break
            else:
                move = current_player.strategy(self.available_moves) # Random plays always X
                self.O_boxes.append(move)
                if self.win_condition(self.X_boxes):
                    winner = opponent
                    print("Uuuups, opponent won")
                    Break

            # Remove the move from the available
            self.available_moves.remove(move)

        if winner == playerT:
            return 1
        elif winner == opponent:
            return -1
        else:
            return 0
```

```
value_dictionary = defaultdict(float)
hit_state = defaultdict(int)
epsilon = 0.05


def update_dict(learner_state, opponent_state, reward):
    hashable_state = (frozenset(learner_state), frozenset(opponent_state))
    hit_state[hashable_state] += 1
    value_dictionary[hashable_state] = value_dictionary[
        hashable_state
    ] + epsilon * (reward - value_dictionary[hashable_state])
```

```
# match = TicTacToe()
playerA = Player("Random1")
playerB = Player("Random2")

for times in range(1_000):
    match = TicTacToe()
    trajectory = match.play_game(playerA, playerB)
    # print(trajectory)

    last_state = trajectory[-1]

    if match.win_condition(last_state.X):
        print("X won")

        for state in trajectory:
            update_dict(state.X, state.O, 1)
            update_dict(state.O, state.X, -1)

    elif match.win_condition(last_state.O):
        print("O won")

        for state in trajectory:
            update_dict(state.O, state.X, 1)
            update_dict(state.X, state.O, 1)
    else:
        print("It is a draw")

        for state in trajectory:
            update_dict(state.X, state.O, 0.5)
            update_dict(state.O, state.X, 0.5)

    print()

print(value_dictionary)
```

```python
playerC = Player("TrainedPlayer")
playerC.set_trained_player()

draws = 0
trained_player_victories = 0
opponent_player_victories = 0

for times in range(1_000):
    match = TicTacToe()

    result = match.play_game_trained(value_dictionary, playerC)

    if result == 1:
        trained_player_victories += 1

    elif result == -1:
        opponent_player_victories += 1
```

```python
    else:
        draws += 1

print(f"Number of matches won by TrainedPlayer: {trained_player_victories}")
print(f"Number of matches won by Opponent: {opponent_player_victories}")
print(f"Number of draw matches : {draws}")
```

FINAL PROJECT

Quixo game

Introduction

The final project requires implementing a player capable of playing **Quixo**. It's a game played on a 5x5 grid of cubes, with each cube having two symbols: a circle and a cross, and empty faces. Players take turns choosing a cube (whether it belongs to them or is unclaimed, with the empty face facing up), rotating it to display their symbol, and then pushing it into the empty square, thereby moving all adjacent cubes in the same direction. If a player creates an alignment of 5 symbols, they win the game.

In the code, the cross and circle are represented by 0 and 1, and the empty face by -1.

My Idea

I've implemented a minimax algorithm: therefore, my player can evaluate all possible moves and choose the most promising one.

To do this, I've developed the recursive function **apply_minimax_algorithm**: during evaluation, the player starts from the current game state and assumes that each player will play in the best way to maximize their chance of winning (and minimize their opponent's).

To assign a value to the considered state, I've used the **evaluate_state function**. It's based on a heuristic I devised, where more points are awarded as the player achieves longer alignments of aligned squares. Specifically, I've assigned 5 points for 2 aligned pieces, 20 for three pieces, 50 for four, and 1000 for five (which equals a win). I've also favored selection on diagonals by giving an additional point if the piece is on one.

Since processing times were very long, I added **Alpha-beta pruning**: this allows discarding less promising branches of the tree.

The function that generates possible moves relies entirely on the checks made within the module provided by the task.

Code (without the game modules)

main.py

```python
import random
from utils import HandleGame
from game import Game, Move, Player
from miniMaxPlayer import MiniMaxPlayer
from tqdm import tqdm


class RandomPlayer(Player):
    def __init__(self) -> None:
        super().__init__()
        self.name = "Random"

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        from_pos = (random.randint(0, 4), random.randint(0, 4))
        move = random.choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
        return from_pos, move
```

```python

if __name__ == '__main__':

    g = Game()
    player1 = MiniMaxPlayer(1)
    player2 = RandomPlayer()
    winner = g.play(player2, player1)
    g.print()
    print(f"Winner: Player {winner}")

    miniMax_wins_first = 0

    with tqdm(total=100, desc="Partite giocate") as pbar:
        for match in range(100):
            g = Game()
            player1 = MiniMaxPlayer(6)
            player2 = RandomPlayer()
            winner = g.play(player1, player2)
            if winner == 0 : miniMax_wins_first += 1

            pbar.update(1)

    print("MiniMax wins", miniMax_wins_first, "matches on 100 (first player to move)")

    miniMax_wins_second = 0

    with tqdm(total=100, desc="Partite giocate") as pbar:
        for match in range(100):
            g = Game()
            player1 = MiniMaxPlayer(14)
            player2 = RandomPlayer()
            winner = g.play(player2, player1)
            if winner == 1 : miniMax_wins_second += 1

            pbar.update(1)

    print("MiniMax wins", miniMax_wins_second, "matches on 100 (second player to move)")
```

miniMaxPlayer.py

```
import random
from utils import HandleGame
from game import Game, Move, Player
from copy import deepcopy
from enum import Enum
import numpy as np

POINTS = [0, 0, 5, 20, 50, 1000]
POINTS_DIAGONAL = [0, 1, 5, 20, 50, 1000]
```

```
class MinOrMax(Enum):
    isMin = -1
    isMax = 1
```

```
class MiniMaxPlayer(Player):
    def __init__(self, max_depth) -> None:
        super().__init__()
        self.max_depth = max_depth
        self.name = "MiniMax"

    def evaluate_state(self, currentGame: 'Game', min_or_max: MinOrMax):

        # Check if the game is over, returning an infinite value coeherent with the min or max
sign
        # if currentGame.check_winner() >=0 : return float("inf")*min_or_max.value
        # The player who made the last move, who is waiting for the value
        player_ID = currentGame.current_player_idx + 1
        player_ID %= 2

        # The current player
        opponent_ID = currentGame.current_player_idx

        # Count the number of pieces in a row and in a column, for each one
        player_pieces_on_row = np.sum(currentGame._board == player_ID, axis=1)
        player_pieces_on_column = np.sum(currentGame._board == player_ID, axis=0)
```

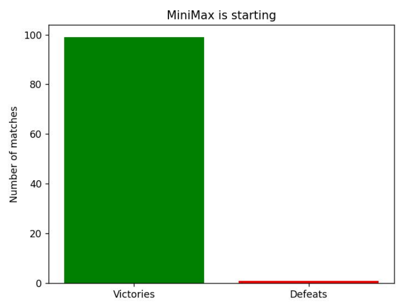
```
        # Count the number of pieces in a row and in a column, for each one
        opponent_pieces_on_row = np.sum(currentGame._board == opponent_ID, axis=1)
        opponent_pieces_on_column = np.sum(currentGame._board == opponent_ID, axis=0)

        # Inizialize the value
        value = 0

        # Assign the score. The longer the sequence the "higher" the score.

        # It is needed to change the sign because this points promote the previuos player
(opposite to the value of the current min_or_max flag)
        for sequence in player_pieces_on_row:
            value += POINTS[sequence]*min_or_max.value*(-1)
        for sequence in player_pieces_on_column:
            value += POINTS[sequence]*min_or_max.value*(-1)

        # Points for the current player
        for sequence in opponent_pieces_on_row:
            value += POINTS[sequence]*min_or_max.value
        for sequence in opponent_pieces_on_column:
            value += POINTS[sequence]*min_or_max.value

        # Count the number of pieces on diagonals
        player_pieces_on_first_diag = np.sum(np.diag(currentGame._board) == player_ID)
        player_pieces_on_second_diag = np.sum(np.diag(np.fliplr(currentGame._board)) ==
player_ID)
        opponent_pieces_on_first_diag = np.sum(np.diag(currentGame._board) == opponent_ID)
        opponent_pieces_on_second_diag = np.sum(np.diag(np.fliplr(currentGame._board)) ==
opponent_ID)

        value += POINTS_DIAGONAL[player_pieces_on_first_diag]*min_or_max.value*(-1)
        value += POINTS_DIAGONAL[player_pieces_on_second_diag]*min_or_max.value*(-1)
        value += POINTS_DIAGONAL[opponent_pieces_on_first_diag]*min_or_max.value
        value += POINTS_DIAGONAL[opponent_pieces_on_second_diag]*min_or_max.value

        return value


    def apply_minimax_algorithm(self, currentGame: 'Game', depth, min_or_max, alpha, beta):

        # End of the iterative calls
        # Check if it arrived in the max depth or the game is over
        if depth == 0 or currentGame.check_winner() >= 0:
            return self.evaluate_state(currentGame, min_or_max), alpha, beta

        # Decrese the current depth
        current_depth = depth - 1

        # Get the ID of the current player
        player_ID = currentGame.get_current_player()
        # Check if the player is the maximising one
        if min_or_max == MinOrMax.isMax :

            # Set the best value as the worst possible
            best_value = float('-inf')






            for move in HandleGame.possible_moves(currentGame):

                # Generate a copy of the game, able to be modified in the iteration
                newGame = deepcopy(currentGame)
                # Apply the current move
                newGame.moves(move[0], move[1], player_ID)
                newGame.current_player_idx += 1
                newGame.current_player_idx %= 2

                # Reiterate the recursive algorithm
                value, alpha, beta = self.apply_minimax_algorithm(newGame, current_depth,
MinOrMax.isMin, alpha, beta)

                # Update the best value
                # best_value = value if value > best_value else best_value
                if value > best_value:
                    best_value = value

                # Handle alpha-beta pruning
                alpha = value if value > alpha else alpha
                if alpha >= beta:
                    break

                if best_value == float("inf"): break

            return best_value, alpha, beta

        # The player is minimising one
        else:

            # Set the best value as the worst possible
            best_value = float('inf')

            for move in HandleGame.possible_moves(currentGame):

                # Generate a copy of the game, able to be modified in the iteration
                # newGame = deepcopy(currentGame)
                newGame = Game()
                newGame._board = deepcopy(currentGame._board)

                # Apply the current move
                newGame.moves(move[0], move[1], player_ID)
                newGame.current_player_idx += 1
                newGame.current_player_idx %= 2

                # Reiterate the recursive algorithm
                value, alpha, beta = self.apply_minimax_algorithm(newGame, current_depth,
MinOrMax.isMax, alpha, beta)
```

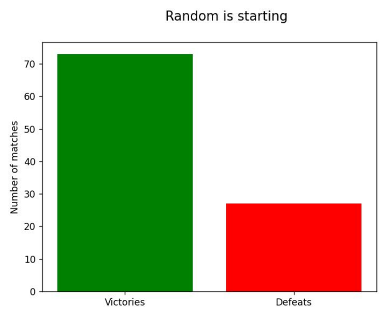
```
                # Update the best value
                # best_value = value if value < best_value else best_value
                if value < best_value:
                    best_value = value

                # Handle alpha-beta pruning
                beta = value if value < beta else beta
                if alpha >= beta:
                    break

                if best_value == float("-inf"): break

            return best_value, alpha, beta


    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:

        # Generate all possible legit moves
        possible_moves = HandleGame.possible_moves(game)
        # Find the best move, by testing all in the following max_depth moves.
        # If the game is not ended in maz_depth moves, it continues randomly to reduce the run
time.

        best_move = None
        best_value = float("-inf")

        # Lookin for the best move
        for move in possible_moves:

            # Generate a copy of the game, able to be modified in the iteration
            newGame = deepcopy(game)

            # Apply the current move
            newGame.moves(move[0], move[1], game.current_player_idx)

            if newGame.check_winner() == newGame.current_player_idx :
                best_move = move
                best_value = float("inf")
                break

            newGame.current_player_idx += 1
            newGame.current_player_idx %= 2

            # Apply the minimax algorithm
            value, alpha, beta = self.apply_minimax_algorithm(newGame, self.max_depth,
MinOrMax.isMin, -float("inf"), float("inf"))

            # Update the results
            # best_value, best_move = (value, move) if value > best_value else (best_value,
best_move)
            if value > best_value:
                best_value = value
                best_move = move

        return best_move
```

utils.py

```python
from copy import deepcopy
from itertools import product
from game import Game, Move
from enum import Enum
import random

# define the corners
SIDES = [(0, 0), (0, 4), (4, 0), (4, 4)]


class HandleGame():

    def possible_moves(currentGame: Game) -> tuple[tuple[int, int], Move]:

        # Copy to be sure it won't be modified
        copyCurrentGame = deepcopy(currentGame)

        # This list will contain all possibile moves
        possible_moves = []

        # Generate all possible combinations of the pieces' positions
        all_combinations = tuple(product(range(5), repeat=2))

        for straight_position in all_combinations:

            position = tuple((straight_position[1], straight_position[0]))

            # Evaluate if the taken piece is legit
            # !!! (rules are taken from the code written in game.py)

            # acceptable only if in border
            acceptable: bool = (
                # check if it is in the first row
                (position[0] == 0 and position[1] < 5)
                # check if it is in the last row
                or (position[0] == 4 and position[1] < 5)
                # check if it is in the first column
                or (position[1] == 0 and position[0] < 5)
                # check if it is in the last column
                or (position[1] == 4 and position[0] < 5)
                # and check if the piece can be moved by the current player
            ) and (copyCurrentGame._board[position] < 0 or copyCurrentGame._board[position] ==
copyCurrentGame.current_player_idx)

            if acceptable:
                # print("Acceptable:", "MiniMaxID", copyCurrentGame.current_player_idx,
"pedina", copyCurrentGame._board[position], "taken at", position)
                # Generate the combination of moves, according to the rules written in game.py

                # if the piece position is not in a corner
                if position not in SIDES:

                    # if it is at the TOP, it can be moved down, left or right
                    if position[0] == 0:
                        possible_moves.append((straight_position, Move.BOTTOM))
                        possible_moves.append((straight_position, Move.LEFT))
                        possible_moves.append((straight_position, Move.RIGHT))

                    # if it is at the BOTTOM, it can be moved up, left or right
                    if position[0] == 4:
                        possible_moves.append((straight_position, Move.TOP))
                        possible_moves.append((straight_position, Move.LEFT))
                        possible_moves.append((straight_position, Move.RIGHT))

                    # if it is on the LEFT, it can be moved up, down or right
                    if position[1] == 0:
                        possible_moves.append((straight_position, Move.BOTTOM))
                        possible_moves.append((straight_position, Move.TOP))
                        possible_moves.append((straight_position, Move.RIGHT))

                    # if it is on the RIGHT, it can be moved up, down or left
                    if position[1] == 4:
                        possible_moves.append((straight_position, Move.BOTTOM))
                        possible_moves.append((straight_position, Move.TOP))
                        possible_moves.append((straight_position, Move.LEFT))

                # if the piece position is in a corner
                else:

                    # if it is in the upper left corner, it can be moved to the right and down
                    if position == (0, 0):
                        possible_moves.append((straight_position, Move.BOTTOM))
                        possible_moves.append((straight_position, Move.RIGHT))

                    # if it is in the lower left corner, it can be moved to the right and up
                    if position == (4, 0):
                        possible_moves.append((straight_position, Move.TOP))
                        possible_moves.append((straight_position, Move.RIGHT))

                    # if it is in the upper right corner, it can be moved to the left and down
                    if position == (0, 4):
                        possible_moves.append((straight_position, Move.BOTTOM))
                        possible_moves.append((straight_position, Move.LEFT))

                    # if it is in the lower right corner, it can be moved to the left and up
                    if position == (4, 4):
                        possible_moves.append((straight_position, Move.TOP))
                        possible_moves.append((straight_position, Move.LEFT))


        # Return a tuple for efficiency
        random.shuffle(possible_moves)
        return tuple(possible_moves)
```

Results





The developed algorithm works well when it starts first. However, we should improve the model for games where it starts second.

Reviews

DONE

Lab 2

Review 1 to Federica000

HI! I chose your lab randomly from the file containing the CI course github links.

Introduction
I'll start by saying that you've created a nice genetic algorithm, quite complex and perhaps for this reason it took me some time to fully understand what the logic of the code was.

If I understand correctly, your goal was to create a population of individuals containing a set of states and a move for each of them. The latest generation should contain sets of optimal actions to carry out in case, during the game, the system ricognizes an already known state. Even if I understand correctly, the knowledge of the algorithm is divided into the various individuals, who are not "independent" but must collaborate to provide an ideal move.

My doubts
The idea is very particular and I liked it a lot, but there are some points that are unclear to me.

In the play function, on the line if state == nim_game: I'm afraid that the comparison will never be successful due to the different representation of the two variables. Running the code it seems to me that it never enters the true condition.

I've the doubt that the algorithm prefers moves that lead to an nim XOR = 0 (which should be exactly the condition to discard). In the last generation, in fact, I saw many moves that, associated with the state, cause an xor equal to zero.

Advice
As advice for a hypothetical future development, it would be interesting to try not to have redundancy within the final generation (therefore managing each state once by choosing only one move).

It might be useful to include a little more comments in the code, just because the algorithm is quite complex.

Conclusion
To conclude, I think you have done an original and different job compared to the others. I really appreciated seeing a different approach.

I hope I have understood your code correctly and if not, I apologize for exposing my doubts to you.

Good luck for the next works, keep it up

Review 1 to Raffaele Viola

HI! I chose your lab randomly from the file containing the CI course github links.

My point of view and suggestions
I'll start by saying that I found the code very readable and well structured.

Since we have some commonalities, I want to share with you some tips (some of which I applied in my own code, some I received from other reviewers)

To avoid training the system on a single challenger (the optimal) you risk having a result with a starting bias. I did the same thing in my code, but it certainly can be interesting to evaluate it through challenges with other individuals and/or strategies.

Even though the code is linear, it might be useful to have some more comments.

I appreciated the progress bar which allows you to see how far along the execution of the algorithm is. However, it would be nice to also print some additional information.

Good luck for the next works, keep it up

DONE

Lab 10

Review 1 to Nicolò Caradonna

Hello Nicolò,
I've conducted a review of your code and wanted to share my observations.

My 2 cents:
Your code is well-structured and organized. I particularly appreciate the clear division into classes, managing both the game logic and player behaviors. This choice significantly enhances the readability and maintainability of the code.

The option to test the trained player in a match against a human player is very interesting. This feature makes your code interactive, providing an immediate test of the achieved results during training.

The integration of graphs at the end of the lab provides a comprehensive view of the learning process. This is a positive touch that offers a visual overview of the agent's performance throughout the training matches.

Recommended Adjustments:

The start_game() function might be considered redundant as it merely prints a static string. You may want to evaluate whether it's essential to keep this function.
The print statements during training could be shortened to enhance file readability. Consider reducing the length of the prints while retaining essential information.

Future Developments:
It could be interesting to explore varying the epsilon variable as the algorithm learns to play. This might help reduce randomness in the agent's moves and reduce exploration during the learning process.
Additionally, it would be compelling to train the agent against players using different strategies. This could provide insights into how well the agent adapts to diverse playing styles.
Overall, you've done an excellent job. Keep it up and consider the suggestions to further enhance your code :)

Review 2 to Michelangelo Caretto

Hello Miki,

I've chosen your code for the peer review.

My 2 cents:
I find your algorithm well-crafted and clear. The logic is easy to follow, contributing to the overall clarity of the code.

I appreciated the fact that, in the final part of the code, you showcase the results obtained by pitting your model against various opponents using different techniques. This provides valuable insights into how well your model performs against diverse strategies.

The visual representation of results in the form of a data table is highly useful. It allows for a clear understanding of the achieved outcomes.

The readme file in the folder is helpful and provides a good introduction for those who wish to study the code you've written.

Recommended Adjustments:
Adding comments within the code would enhance its comprehensibility. Consider including comments to explain specific sections or functions.

Improving code readability by inserting spaces between different logical blocks within the same function would be beneficial.

Future Developments:
It could be interesting to train the model against specific strategies or combinations of strategies, rather than solely random players. For instance, training it against known strategies while leaving some opponents with "unknown" strategies could provide a more accurate final assessment, as it would face adversaries it has never encountered.

Overall, you've done a great job. Keep it up and consider the suggestions to further enhance your code :)

RECEIVED

Lab 2

Review by Caretto Michelangelo s310178

Hello Angelo,
i'm going to review your work , hope you'll enjoy.
I like a lot the way you wrote the code, because is so clean and readable and due to the graph i can easily understand how your "weight" structure works and which strategy is more strong in term of fitness.
I would suggest you , when trying to create an agent playing a game to switch starting player every match too,otherwise your training will be only by one starting side.

Talkink about the Es algortihm , you managed to find a solution in 25 Generation ,due to the fact of "Optimal function" is not optimal , but in general we can not call an algortihm "ES" with this low number of Generation .
Next time maybe with a little curiosity you could study the Nim problem more to make the optimal function stronger.... (less homework done and more curiosity)
I would like to say that despite everything, I appreciated the work and the cleanliness.

Thank you for your effort and attention to detail.
Bye Bye Angelo,
Michelangelo.

Review by Alexandro Buffa S316999

Hi Angelo, you've been picked randomly from random.org for my peer review, nothing personal.

Your code is very well structured, well commented and very straight-forward to read and understand.
I also liked your approach on trying to find new strategies to compete against the optimal strategy, and thanks to your graphs it's easy to see that your results look promising.

The only thing left for me to add is that you are using a 1+lambda approach (instead of the (1, lambda) noted above the code), since you are appending the parent to the offspring and then picking the best individual (which could be the parent of the previous generation). This is not a problem per se, but it might be interesting to see how the results change with a 1, lambda approach.

Another twist that could spice things up is to try and train the individuals by playing different versions of the game (different sizes, with/without k--max pieces nimmable) and by playing different sides (first/second player).

That said, I think you did a great job with this lab!

Review by Alessio Cappello s309450

Hi Angelo, I looked at your code and must say it's well written.

I appreciated the graphs showing the trends of the values belonging to the genome, it helped me understand your idea better. Nevertheless, personally, I would have liked to see something more personal than only taking some strategies well-defined "weighted randomly": it could happen that you're using always the same strategy (i.e. the "longest"). Furthermore, the strategy is 1+λ and not 1,λ since you're also considering the parent as a candidate.

The idea is okay, but maybe using that in different ways would've been better.

Keep working that way and writing code easy to understand, I'm sure you can reach impressive results.
Good luck for next labs! :)

Review by Claudio Macaluso s317149

Hello Angelo,

Great work on the code! I found your approach really commendable for its simplicity and effectiveness. The additional plots in the notebook significantly enhance the system's evolution visualization. I might borrow your idea for future projects if that's okay with you! >:)

For future implementations, here are some areas to consider:

Using only one strategy for opponent fitness evaluation might introduce bias. Experimenting with different strategies could yield fascinating conclusions (e.g., checking if results converge to the same strategy).
Exploring how random or varying initialization affects the final results could be an interesting avenue to pursue.
Just a note: Your strategy is (1+lambda), not (1,lambda), as you're considering the parent as a candidate solution as well, rather than discarding it.
Keep up the great work!

Review by Loris Fabrizio Mario Vitale s316264

I took a dive into your work implementation for the Nim game, and here are my thoughts:

As a first thing appreciate the fact that the code is very clean and modular, with minor duplications, and enough comments to understand the logic flow, but I think it would be better describing more in the depth the overall code on the readme.

The EA that weights the strategy to adopt based on the genome sounds like a good idea.
The results indeed showcase a clear evolution towards strategies over generations adjusting the sigma parameter as it goes trough, but in both the strategies (+,

comma) seems like to favorite always the longest one, this could be due to the fact that the optimal startegy is not actually optimal because it behaves like a random strategy (and tends to lose) during the most important phase of the match, the "end game", that is when, among the rows only one have more than one element remaining, so this makes me think that the algorithm actually evolves the strategy to shortening the game and strike where the optimal is weak (it's just a guess it makes sense)

It would be interesting trying to exploit different initial solutions (and also a real optimal between the possible strategies to adopt) to see what happens and let the algorithm run for more generation (25 seems to few).

The use of matplot is a nice touch, it motivates me to learn it to improve my next works.

Overall, it's a well-crafted implementation with a good balance between complexity and readability. Keep up the great work!

RECEIVED

Lab 3

Review by Davide Natale

Hello there ,
Thanks for sharing your code with the community.
While I appreciate your effort to use the 'islands' concept in your algorithm, I think you had some trouble understanding the exercise requirements.

First of all, the README doesn't explain the whole process you went through to write the code or why you chose certain parameters.

Also, I don't see any attempt to test different parameters, and there's no mention of it in the README, so I assume it wasn't done (but I might be wrong).

The output is too long and I found it hard to read. Plus, I don't see the point of plotting generations along with their fitness values. The main goal of the lab wasn't just to maximize fitness coverage but to find a balance between the total fitness calls and fitness coverage. So, if you already have 99.0% fitness in generation 393, why continue for hundreads of generations with an extra 30k fitness calls for just a 1% improvement?

You also didn't test different problem instances (1, 2, 5, 10) as the exercise required; you only tested instance 1.

That being said, I think you could have done a better job but it still is a solution! Good luck for the next labs!

RECEIVED

Lab 9

Review by Davide Natale

Your code is well-structured and understandable, I liked your attention to add comments which are helpful to understand your implementation, but I think you should have removed or at least reduced the amount of output in your code, this would have given the code much more clarity. I apprecciate your attempt to promote diversity by implementing the island model.
I didn't fully understand the results you showed, in fact I think you didn't test all the Problem instances.
Anyway, good job!