1. Introduction

This report contains all the details on the implementation of the various laboratory activities done in the Computational Intelligence course during the academic year 2023/2024.

2. Lab 1: Set Covering

The first lab covered is about the set covering problem. The problem consists in a finite number of elements that we can cover with a finite number of sets. Each set covers different elements. The goal is to find the minimum number of sets required to cover all the elements. The solution is implemented with the A* algorithm. The algorithm searches for the path that minimizes the following function:

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

where n is the next node on the path, g(n) is the actual cost of the path from the starting node and h(n) is the estimated cost to get from n to the goal state. This last cost is computed through a heuristic function. Such function needs to be admissible, which means that it never overestimates the cost. For g(n), since our goal is to minimize the number of sets used, I decided to use the number of sets used to reach the node n. As for h(n), I tried different heuristics and confronted their results. The first function tried computes the number of elements who are still not covered, reorders all the non-used sets based on the number of non-covered elements they manage to cover and then takes the minimum number of ordered sets that have a number of covered elements greater or equal to number of elements still needing to be covered.

```python
def compute_min_tiles(state):   #look how many blocks are still not filled and estimate the minimum number of tiles required to fill them
    covered = coverage(state)
    missing = PROBLEM_SIZE - sum(covered)
    sort_cand = sorted(sum(np.logical_and(SETS[c], np.logical_not(covered)) for c in state.not_taken), reverse=True)
    taken = 1
    while sum(sort_cand[:taken]) < missing:
        taken += 1
    return taken
```

The second function is very similar to the first one, but this time it stops when all the elements that must be covered are really covered.

```python
def compute_min_tiles2(state):   #this actually computes how many tiles are required to fill the whole space, but it's not an estimation
    added = 0
    covered = coverage(state)
    while not np.all(covered):
        # missing = PROBLEM_SIZE - sum(covered)
        best_cand = max(state.not_taken, key=lambda c : sum(np.logical_and(SETS[c], np.logical_not(covered))))
        state = State(
            state.taken ^ {best_cand},
            state.not_taken ^ {best_cand},
        )
        added += 1
        covered = np.logical_or(covered, SETS[best_cand])
    return added
```

Another function tried consists in computing the minimum number of required sets by adding the set that covers the most uncovered elements one at a time. This is not really an estimation though and it's also highly inefficient. The 2 functions tested are very similar and the main idea is to sort the non-used sets based on their diversity from the coverage of the current state. The first one computes the diversity based on the number of elements of the coverage state that are not also covered by the new set, whereas the second one computes it by counting the number of elements that are covered in the state coverage and not covered by the new set and vice versa.

```python
def compute_min_tiles3(state):  #sort the remaining tiles based on the amount of non overlapping blocks
    added = 0
    covered = coverage(state)
    sort_cand = sorted(state.not_taken, key=lambda c : sum(np.logical_and(SETS[c], covered)))
    for c in sort_cand:
        if np.all(covered):
            break
        covered = np.logical_or(covered, SETS[c])
        added += 1
    return added


def compute_min_tiles4(state): #similar to the previous one, the idea is to select the tiles that are filling non-filled blocks and are not filling
    added = 0
    covered = coverage(state)
    sort_cand = sorted(state.not_taken, key=lambda c : sum(np.logical_xor(SETS[c], covered)), reverse=True)
    for c in sort_cand:
        if np.all(covered):
            break
        covered = np.logical_or(covered, SETS[c])
        added += 1
    return added
```

This last one proved to be the most effective function to use for our heuristic, giving a solution of 4 tiles required to cover 50 elements with 100 sets at our disposal. It also allows to find the solution very fast.

```python
frontier.put((len(new_state.taken) + compute_min_tiles4(new_state), new_state))
```

This is the function I used to insert in the frontier (implemented as a priority queue) the new state created with its relative priority given by its cost and by the value of the heuristic function. Below you can see the implementation of the whole algorithm including the initialization of the frontier and the termination condition.

```python
frontier.put((distance(state), state))

counter = 0
skip_count = 0
_, current_state = frontier.get()

while not goal_check(current_state):
    if sorted(current_state.taken) not in already_explored:
        already_explored.append(sorted(current_state.taken))
        counter += 1
        for action in current_state[1]:
            new_state = State(
                current_state.taken ^ {action},
                current_state.not_taken ^ {action},
            )

            frontier.put((len(new_state.taken) + compute_min_tiles4(new_state), new_state))

    else:
        skip_count += 1
    _, current_state = frontier.get()


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

We can notice the presence of an already_explored variable. This is meant to be a list of all the nodes we already analyzed. It was mostly used to check if we ever run into an already explored node, but with the last heuristic used this never happens.

3. Halloween Challenge

The Halloween challenge consisted in trying to solve the set covering problem with hill climber while trying to keep the number of calls to the fitness function low. The hill climber is a single state method and it's based on constantly improving the current solution by tweaking it until a local maximum is reached. We can view the objective function as a hill and the agent's purpose is to find the solution that maximizes such function, or in other words, gets him to the top of the hill. In this challenge, I tried different variations of the hill climber algorithm. But first the tweak function and the fitness must be defined.

```
def tweak(state):
    new_state = copy(state)
    i = randint(0, len(state) - 1)
    new_state[i] = not new_state[i]
    return new_state
```

```
def fitness1(state):
    cost = sum(state)
    filled = np.sum(
        reduce(
            np.logical_or,
            [x[[i], :].toarray() for i, t in enumerate(state) if t],
            np.array([False for _ in range(len(state))]),
        )
    )
    return filled, -cost
```

The tweak function is simply a random modification of a single value of the state, which is composed by a list of True/False values (if state[i] is True than the set number i is considered used in the current state). The fitness function instead returns the number of covered elements in the input state and also the number of sets used (with a minus sign because the fitness value wants to be as high as possible but of course if less sets are used to get the same number of covered elements than said solution is better).

The first variation is the First Improvement variation, which is the simplest implementation of the hill climber. This variant of the algorithm simply tweaks the state for a given number of times and if the new state has a better fitness value than this solution than on the next iteration that new state will be the new solution it'll try to optimize. To avoid a great number of iterations, I decided to implement the steady state function. This consists if analyzing after every tweak if it ends up getting a better solution and if that doesn't happen for many tweaks in a row than you are probably already in a local or global optimum and so the current solution is returned without tweaking it any longer. Below you can see the implementation of the First Improvement variation.

```python
def first_improvement(state, fitness):
    s =  state.copy()
    fit_s = fitness(s)
    n_calls = 1
    steady = 0
    for step in range(5000):
        r = tweak(s.copy())
        fit_r = fitness(r)
        n_calls += 1
        if fit_r > fit_s:
            s = r
            fit_s = fit_r
            steady = 0
        else:
            steady += 1
        if steady == 500:
            print(f"Interruption at the {step}-th step")
            break
    return s, n_calls
```

The next variation I tried is the Steepest Step Hill Climber, which is almost equivalent to the previous variation with the only difference that this time we tweak the current best solution many times per iteration, than we select the best solution obtained from all the tweaks and if said solution has a higher fitness value of the current best solution, than that solution will be tweaked moving forward instead.

```python
def steepest_step_hc(state, n_samples, fitness):
    s = state.copy()
    fit_s = fitness(s)
    steady = 0
    n_calls = 1
    for step in range(5000):
        r = tweak(s.copy())
        fit_r = fitness(r)
        n_calls += 1
        for _ in range(n_samples - 1):
            w = tweak(s.copy())
            fit_w = fitness(w)
            n_calls += 1
            if fit_w > fit_r:
                r = w
                fit_r = fit_w
        if fit_r > fit_s:
            s = r
            fit_s = fit_r
            steady = 0
        else:
            steady += 1
        if steady == 500:
            print(f"Interruption at the {step}-th step")
            break
    return s, n_calls
```

The penultimate variation that I tested was the Simulated Annealing algorithm. This variant consists of tweaking the current best solution and if the state obtained through the tweak was a better solution than it would replace the old one as the current best solution. But this time there is a chance that it could replace it anyway even if it has a lower fitness value. This is implemented with the introduction of a temperature value, which decreases at every tweak. The higher the temperature, the higher the chance of accepting a worse solution. In this way we can encourage exploration in the early iterations of the algorithm and then switch to exploitation in the last iterations. I decided to adopt the linear schedule for the temperature decreasing rate because it allows the temperature value to drop faster than it would with other kinds of schedule (such as the logarithmic one, which I also implemented), otherwise I would have needed a bigger number of iterations therefore increasing the number of fitness calls.

```python
def fit_diff(fit1, fit2):
    return fit1[0] + fit1[1] - fit2[0] - fit2[1]

def log_schedule(init_temp, step):
    return init_temp / (1 + log(1 + step))

def lin_schedule(init_temp, step):
    return init_temp / (1 + step)
```

The schedule parameter of the Simulated Annealing function corresponds to the lin_schedule() function

```python
def simulated_annealing(state, init_temp, schedule, fitness):
    s = state.copy()
    fit_s = fitness(s)
    steady = 0
    num_calls = 1
    for step in range(5000):
        r = tweak(s.copy())
        fit_r = fitness(r)
        num_calls += 1
        if fit_r > fit_s:    #tweaked state is better
            s = r
            fit_s = fit_r
            steady = 0
        else:     #do I accept the worse solution?
            t = schedule(init_temp, step)
            p = exp(-fit_diff(fit_s, fit_r)/t)
            # print(f"prob di accettare: {p}")
            if random() <= p:
                s = r
                fit_s = fit_r
                steady = 0
            else:
                steady += 1
        if steady == 500:
            print(f"Interruption at the {step}-th step")
            break
    return s, num_calls
```

And then there is the last version of the Hill Climber I tried, which is the Tabu Search. My implementation of this variant is very similar to the Steepest Step algorithm, but this time there is a Tabu list used to store all the last observed states. If a tweak gives a result a state which is already present in said list, then the candidate is discarded, and its fitness value is not computed. If the state is not included in the list yet, then we first compute its fitness value and only after this we add it to the tabu list.

```python
def tabu_search(state, n_samples, fitness):
    max_size = 500
    best_cand = state.copy()
    best = state.copy()
    steady = 0
    tabu_list = []
    tabu_list.append(best_cand)
    num_calls = 0
    for step in range(5000):
        fit_best_cand = (-inf, -inf)
        tmp = [tweak(best_cand) for _ in range(n_samples)]
        cand = [sol for sol in tmp if not check_tabu_presence(sol, tabu_list)]
        for c in cand:
            f = fitness(c)
            num_calls += 1
            if f > fit_best_cand:
                best_cand = c
                fit_best_cand = f
        if fit_best_cand == (-inf, -inf):
            break
        fb = fitness(best)
        num_calls += 1
        if fit_best_cand > fb:
            fb = fit_best_cand
            best = best_cand
            steady = 0
        else:
            steady += 1
        if steady == 500:
            print(f"Stop at the {step}-th step")
            break
        tabu_list.append(best_cand)
        if len(tabu_list) > max_size:
            tabu_list.pop(0)

    return best, num_calls
```

In the end, the best results obtained are the following:

Num Points	Density	Solution Size	# of fitness calls	Algorithm
100	0.3	7	1483	Simulated Annealing
1000	0.3	14	846	Simulated Annealing
5000	0.3	22	525	First Improvement
100	0.7	3	648	Simulated Annealing
1000	0.7	5	1507	Simulated Annealing
5000	0.7	7	594	First Improvement

4. Lab 2

The goal of lab 2 was to create an agent able to play the nim game. The game consists in a board composes by various rows, each one having one or more matches. On each turn, a player can take any number of matches from 1 to k from a row of his choice, where k is either the number of matches of the row he chose or a predefined number. The player who performs the last move loses the game. The optimal strategy consists in performing moves that leave the board in a situation such as that the nim-sum of the number of matches belonging to every single row returns a result different from 0. Our goal is to find an alternative strategy to create a player able to compete against a player who plays following the optimal strategy just mentioned. Specifically, we want to define an alternative strategy using an ES strategy. I decided to adopt the $\mu + \lambda$ strategy so that I could preserve the best individuals independently from their generation. First, we need to define a phenotype.

```python
for _ in range(POPULATION_SIZE):   #initialize population
    init_values = list(np.random.dirichlet(np.ones(3)))
    new_ind = {
        "rules": {
            "emptiest_row": init_values[0],
            "fullest_row": init_values[1],
            "largest_take": init_values[2],
        },
        "variance": 1
    }
    n_wins = play_many_matches(new_ind, opponent_strategy, TRAIN_MATCHES)
    population.append((new_ind, n_wins))
```

As we can see, every member of the initial population is initialized with 3 different probabilities, each of which represents the possibility a given individual has of performing a specific move. The 3 kinds of moves I have chosen are:

- Emptiest row: perform the move that minimizes the number of matches still to take off from the row where you perform the move.
- Fullest row: perform the move that maximizes the number of matches still to take off from the row where you perform the move.
- Largest take: perform the move that allows you to take as many matches as possible.

In addition to that, every individual has a variance which will be used to generate the offspring for the next generation.

Now, we need to define a fitness value to determine how good every individual is. In this case the fitness function simply consists in letting every individual play 100 matches against the optimal player and memorizing the number of wins. It corresponds to the play_many_matches() function at the end of the picture above. Its implementation is the following:

```python
def play_many_matches(genome: dict, opponent_strategy, n_matches):
    n_wins = 0
    for _ in range(n_matches):
        if play_single_match(genome, opponent_strategy) == 0:
            n_wins += 1
    return n_wins
```

```python
def play_single_match(genome: dict, opponent_strategy):
    strategy = (None, opponent_strategy)

    nim = Nim(N_ROWS, K)
    # logging.info(f"init : {nim}")
    player = 0
    while nim:
        if strategy[player] is not None:
            ply = strategy[player](nim)
        else:
            moves = find_all_moves(nim)
            move = choose_play(genome)
            ply = make_move(move, nim, moves)
        # logging.info(f"ply: player {player} plays {ply}")
        nim.nimming(ply)
        # logging.info(f"status: {nim}")
        player = 1 - player
    # logging.info(f"status: Player {player} won!")
    return player
```

The optimal opponent selects all the moves which return a nim-sum different from 0, whereas the agent researches all the possible moves he can do, then he chooses the rule to apply among the 3 he has (the sum of the 3 probabilities will always be equal to 1) and then performs the move according to the rule he chose to apply.

```python
def choose_play(gen: dict):
    return np.random.choice(list(gen["rules"].keys()), p=list(gen["rules"].values()))

def make_move(mov, nim: Nim, spicy_moves: list):
    best_sol = None
    rows = nim.rows
    if mov == "emptiest_row":   #find the move that leaves its own row as empty as poss
        for sp in spicy_moves:
            rem_elem = rows[sp[0]] - sp[1]
            if best_sol == None or rem_elem < best_sol[1]:
                best_sol = (sp, rem_elem)
    elif mov == "fullest_row":   #find the move that leaves its own row as full as poss
        for sp in spicy_moves:
            rem_elem = rows[sp[0]] - sp[1]
            if best_sol == None or rem_elem > best_sol[1]:
                best_sol = (sp, rem_elem)
    elif mov == "largest_take":   #find the move that allows you to take the greatest p
        for sp in spicy_moves:
            if best_sol == None or sp[1] > best_sol[1]:
                best_sol = (sp, sp[1])
    return best_sol[0]
```

All that remains to be done now is to define how the individuals of the various generations are created. Like previously stated, I opted for the $\mu + \lambda$ strategy.

```
for gen in tqdm(range(N_GENERATIONS)):
    offspring = []
    for ind in population:
        ind[0]["variance"] = np.abs(np.random.normal(loc=ind[0]["variance"], scale=0.2))   #modifying the variance of the individual
        for _ in range(OFFSPRING_SIZE // POPULATION_SIZE):
            new_values = normalize_values(np.abs(np.random.normal(loc = list(ind[0]["rules"].values()), scale = ind[0]["variance"])))
            k = list(ind[0]["rules"].keys())
            new_ind = dict()
            new_ind["rules"] = dict()
            new_ind["variance"] = ind[0]["variance"]
            for i in range(len(k)):
                new_ind["rules"][k[i]] = new_values[i]
            count_wins = play_many_matches(new_ind, opponent_strategy, TRAIN_MATCHES)
            offspring.append((new_ind, count_wins))

    population += offspring

    population = sorted(population, reverse=True, key=lambda e: e[1])
    population = population[0:POPULATION_SIZE]

    if population[0][1] > TRAIN_MATCHES / 2:
        print(f"Stopped at the {gen + 1}-th generation")
        break

return max(population, key=lambda e : e[1])
```

For every generation, we first modify the variance of all the individuals, then each one will be used to generate one or more individuals (depending on the values of μ and λ). I applied a gaussian noise to their probabilities adopting as average value the current values of their probabilities and as variance the value stored in the individual. The newly computed values than get normalized so that their sum will be equal to 1. At this point, the new individual is created, and the fitness function is applied to it. At the end of each generation, I keep the best μ individuals (again, since it's μ + λ, independently from the generation they belong to). After all the iterations are done, only the best individual is kept, and I also make it play 200 more matches against the optimal opponent to furtherly value how good its performance is.

```
count_wins = play_many_matches(winner[0], optimal, TEST_MATCHES)
print(f"Final win rate: {count_wins}/{TEST_MATCHES}")
```

In my case, the best individual I could get was able to win 79 games out of 200.

5. Lab 9

This lab consisted in trying to solve a problem regarding a list of 0/1 values. The fitness function consisted into dividing the list into x sub-vectors (with x = 1, 2, 5, or 10), summing the values of every sub-vector, adding together all the values equal to the maximum, and then subtracting f * 10^{-k} for every value that wasn't equal to the max.

```
@staticmethod
def onemax(genome):
    return sum(bool(g) for g in genome)

def __call__(self, genome):
    self._calls += 1
    fitnesses = sorted((AbstractProblem.onemax(genome[s :: self.x]) for s in range(self.x)), reverse=True)
    val = sum(f for f in fitnesses if f == fitnesses[0]) - sum(
        f * (0.1 ** (k + 1)) for k, f in enumerate(f for f in fitnesses if f < fitnesses[0])
    )
    return val / len(genome)
```

First, I needed to establish what an individual is composed of. In this case, they are just identified by their sequence of 0/1 values, which constitutes their genotype. After this, I decided to adopt the

island model. I divided the entire population in 4 islands and allowed the various populations to swap some of their individuals every 10 generations to try and promote diversity. Their following code shows how the initial population is created and how next generations are handled.

```
population = []  #every element will correspond to the population of an island
fit_history = []
fit_history_per_island = [[] for _ in range(N_ISLANDS)]
generations = []

for island in range(N_ISLANDS):
    p = []
    for _ in range(POP_SIZE):
        ind = choices([0, 1], k=1000)
        p.append((ind, fitness(ind)))
    population.append(p)
```

```
for gen in tqdm(range(NUM_GEN)):

    for island in range(N_ISLANDS):
        parents = []
        parents.append(tournament_selection(population[island].copy()))
        parents.append(tournament_selection(population[island].copy()))

        # parents = tournament_selection(population[island].copy())

        off = []
        c1, c2 = crossover(parents[0][0].copy(), parents[1][0].copy())

        avg_fit = np.average([e[1] for e in population[island]])
        count = 0
        for e in population[island]:
            if e[1] < avg_fit:
                count += 1
        if count >= POP_SIZE // 2:
            mut_prob = POP_SIZE / (count * len(c1))
        else:
            mut_prob = 1 / len(c1)
        c1 = mutation(c1, mut_prob)
        c2 = mutation(c2, mut_prob)
        off.append((c1,fitness(c1)))
        off.append((c2, fitness(c2)))

        population[island] = sorted(population[island] + off, key=lambda e: e[1], reverse=True)[0:POP_SIZE]
```

```
    if (gen + 1) % MIG_STEP == 0:  #perform the migration
        isl = np.random.choice(range(len(population)), size=2, replace=False) # select 2 islands to perform the swap
        population[isl[0]], population[isl[1]] = migration(population[isl[0]], population[isl[1]])

    for i in range(N_ISLANDS):
        fit_history_per_island[i].append(population[i][0][1])

    # for i in range(len(population)):
    #     if check_inequality(population[i]) == False:
    #         print(f"{i + 1}-th island is fully equal at {gen + 1}-th iteration")

    top = []
    for p in population:
        top.append(p[0])
    best = max(top, key=lambda e: e[1])
    generations.append(gen + 1)
    fit_history.append(best[1])

    if best[1] == 1:
        break

print(f"Best individual: {''.join(str(g) for g in best[0])}")
print(f"Best fitness: {best[1]:.2%}")
print(f"Number of fitness calls: {fitness.calls}")
print(f"Number of mutations: {NUM_MUT}")
```

Now there are still a few details to clarify. The first thing is how the parent selection is decided. I have tried different alternatives and I eventually opted for a tournament selection where we select the fittest individuals from a random pool.

```
def tournament_selection3(pop): # select the fittest individual
    participants = np.random.choice(range(len(pop)), size = TOURNAMENT_SIZE, replace=False)
    parents = []
    for p1 in participants:
        parents.append((p1, pop[p1][1])) #(index of the individual, fitness)
    winner = max(parents, key=lambda e: e[1])[0]
    return pop[winner]
```

The next thing that needs to be defined is how recombination is performed. Again, after different implementations, I decided to use the 2-point crossover.

```
def two_point_crossover(p1, p2):
    c = np.random.randint(0, len(p1))
    d = np.random.randint(0, len(p2))

    if d < c:
        swap = c
        c = d
        d = swap

    for i in range(c, d + 1, 1):
        swap = p1[i]
        p1[i] = p2[i]
        p2[i] = swap

    return p1, p2
```

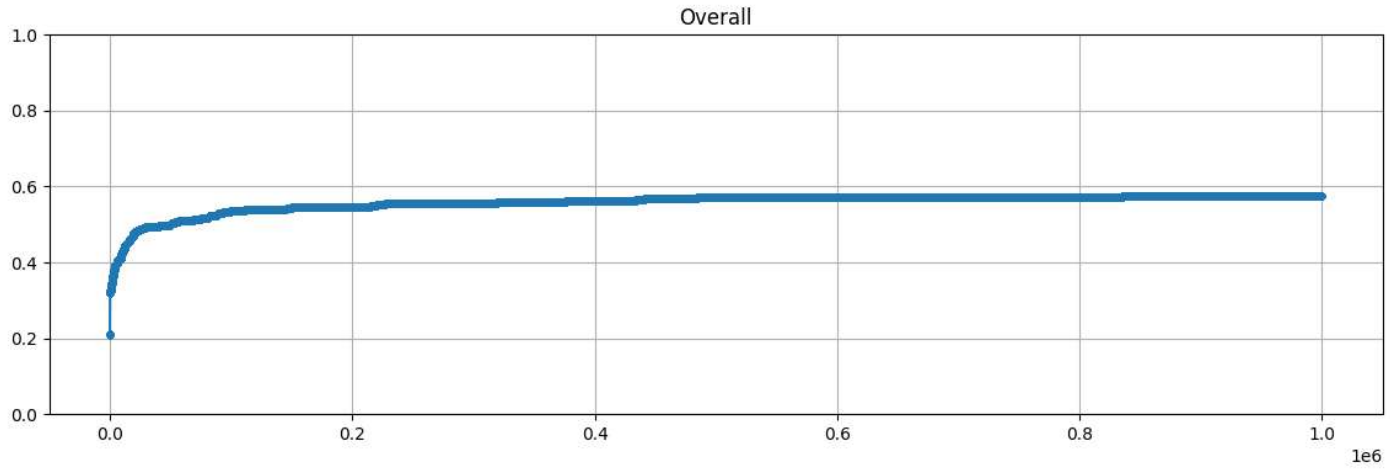
Finally, we still need to define how to handle the migration between the islands. As shown in the code above, I first select 2 random islands which are going to swap a few of their individuals. Then, for each island I sort all the individuals according to how different they are from the individuals (using the Hamming distance) of the other island and then I swap the individuals which are the most different to ensure that the migration provides the best possible outcome in terms of diversity of the population.

```python
def migration2(pop1, pop2):
    # Before swapping the individuals between the 2 chosen islands,
    # we search for the individuals of an island that are the most different from the individuals of the other island to ensure
    # that the swap provides a benefit in terms of diversity

    p1_difference = []  # These lists are used to keep track of the differences
    p2_difference = []

    for i1 in range(len(pop1)):  # Computes the differences for each individual
        p1_difference.append((i1, compute_diff(pop1[i1], pop2)))
    for i2 in range(len(pop2)):
        p2_difference.append((i2, compute_diff(pop2[i2], pop1)))

    e1 = sorted(p1_difference, key=lambda e: e[1], reverse=True)[0:ELEMENTS_SWAPPED] # Retrieves the indexes of the 2 individuals we want to swap
    e2 = sorted(p2_difference, key=lambda e: e[1], reverse=True)[0:ELEMENTS_SWAPPED]

    for i in range(ELEMENTS_SWAPPED):
        swap = pop1[e1[i][0]]
        pop1[e1[i][0]] = pop2[e2[i][0]]
        pop2[e2[i][0]] = swap
    # swap = pop1[e1]  # Swap
    # pop1[e1] = pop2[e2]
    # pop2[e2] = swap

    return sorted(pop1, key=lambda e: e[1], reverse= True), sorted(pop2, key=lambda e: e[1], reverse=True)
```

```python
def compute_diff(ind, pop):  # computes the complexive hamming distance between an individual and an entire population
    diff = 0
    for p in pop:
        diff += hamming_dist(ind[0], p[0])
    return diff
```

```python
def hamming_dist(ind1, ind2):
    return sum(np.array(ind1) != np.array(ind2))
```

Now that I have defined all the details of the strategy I adopted, let's look at the results.



The following graphic is showing the best fitness value for the entire population after every generation. There is a decently fast rise of the value in the first generations but then begins to converge to 0.6. The problem is solved when the fitness value is equal to 1. This is the graphic for x=10. For x=1 and x=2, the problem was easily solvable with my strategy, but it wasn't as easy for x=5 already.

6. Lab 10

The purpose of the last lab was to train an agent to play tic-tac-toe using reinforcement learning. I decided to use the model-free Q-learning algorithm. The details of this algorithm are the following:

Algorithm 124 *Model-Free Q-Learning*

$\alpha \leftarrow$ learning rate ▷ $0 < \alpha < 1$. Make it small.
$\gamma \leftarrow$ cut-down constant ▷ $0 < \gamma < 1$. 0.5 is fine.

$Q(S, A) \leftarrow$ table of utility values for all $s \in S$ and $a \in A$, initially all zero
repeat
    Start the agent at an initial state $s \leftarrow s_0$ ▷ It's best if s_0 isn't the same each time.
    repeat
        Watch the agent make action a, transition to new state s', and receive reward r
        $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$
        $s \leftarrow s'$
    until the agent's life is over
until $Q(S, A)$ isn't changing much any more, or we have run out of time
return $Q(S, A)$ ▷ As our approximation of $Q^*(S, A)$

First, we define a Tic-Tac-Toe class with the following methods:

```python
def __init__(self):
    self.board = np.array([[1, 6, 5], [8, 4, 0], [3, 2, 7]])
    self.available_moves = list(range(9))
    self.current_player = "X"
```

```python
def play_game(self, players: list[Player], train=True, my_player=0):
    current_player = 0
    n_moves = 0
    if my_player == 1:
        n_moves += 1
        init_state = get_state(sorted(players[0].moves), sorted(players[1].moves)) # state before the player performs the move
        move = players[current_player].make_move(self.available_moves)
        self.available_moves.remove(move)
        current_player = 1
    while len(self.available_moves) > 0:
        n_moves += 1
        init_state = get_state(sorted(players[0].moves), sorted(players[1].moves)) # state before the player performs the move
        move = players[current_player].make_move(self.available_moves)
        self.available_moves.remove(move)
        next_state = get_state(sorted(players[0].moves), sorted(players[1].moves)) # state after the player has done its move
        reward = 1 if self.check_winner(players[current_player].moves) else 0
        # print(f"#{n_moves}, init_state: {init_state}, move: {move}")
        if reward == 1:
            if train:
                Q[init_state][str(move)] = (1 - LEARNING_RATE) * Q[init_state][str(move)] + LEARNING_RATE * (reward + CUT_DOWN * max(Q[next_state].values(), default=0))
            return current_player

        if len(self.available_moves) == 0:
            Q[init_state][str(move)] = (1 - LEARNING_RATE) * Q[init_state][str(move)] + LEARNING_RATE * (reward + CUT_DOWN * max(Q[next_state].values(), default=0)) # re
            return None

        current_player = 1 - current_player
        n_moves += 1
        move = players[current_player].make_move(self.available_moves)
        self.available_moves.remove(move)
        reward = -1 if self.check_winner(players[current_player].moves) else 0
        if train:
            # print(f"#{n_moves}, init_state: {next_state}, move: {move}")
            Q[init_state][str(move)] = (1 - LEARNING_RATE) * Q[init_state][str(move)] + LEARNING_RATE * (reward + CUT_DOWN * max(Q[next_state].values(), default=0))
```

```python
        if train:
            # print(f"#{n_moves}, init_state: {next_state}, move: {move}")
            Q[init_state][str(move)] = (1 - LEARNING_RATE) * Q[init_state][str(move)] + LEARNING_RATE * (reward + CUT_DOWN * max(Q[next_state].values(), default=0))
        if reward == -1:
            return current_player
        current_player = 1 - current_player

    return None
```

```
def check_winner(self, player_moves):  # checks if a player has won the game
    if sum(sum(h) == 12 for h in permutations(player_moves, 3)):
        return True
    return False
```

The board is realized with a magic square, which has the peculiarity of having the sum of the cells belonging to the same rows, columns and diagonals equal to the same value, which is 12 in this case.

Then there 2 more classes, which are shown below, used for the players. The first type only keeps the list of moves he has done and chooses his moves randomly among the available ones. The last one instead also takes the already built Q-table in its constructor and performs the moves finding the entry of the beginning state in the Q-table and choosing the move with the highest reward.

```
class Player():
    def __init__(self):
        self.moves = []

    def make_move(self, available_moves):
        move = choice(available_moves)
        self.moves.append(move)
        return move

class MyPlayer(Player):
    def __init__(self, Q):
        super().__init__()
        self.Q = Q

    def make_move(self, available_moves: list[int]):
        # first I need to see which moves are already made by the opponent
        opponent_moves = list(set(list(range(9))) - set(available_moves) - set(self.moves))
        state = get_state(sorted(self.moves), sorted(opponent_moves))

        move = max(available_moves, key = lambda e: self.Q[state][str(e)])
        self.moves.append(move)
        return move
```

Now that the Q-table is complete, we test our player against an opponent who plays randomly.

```python
n_wins = 0
n_draws = 0
n_losses = 0

for _ in range(100):
    mp = MyPlayer(Q)
    opp = Player()
    g = TicTacToe()
    winner = g.play_game([mp, opp], train=False, my_player=0)
    if winner is None:
        n_draws += 1
    elif winner == 0:
        n_wins += 1
    else:
        n_losses += 1
print(f"{n_wins}, {n_draws}, {n_losses}")
```

My player has won 57 times, tied 34 times and lost only 9 times. So, when starting the game, he proved to have a decent chance of winning and a very low chance of losing. But what if the opponent has the first move? The results are completely opposite because of an error in the training code that assigns the high rewards when the first player wins, no matter who said player is.

7. Peer Reviews

Some of the lab activities also included peer reviews. This allowed me and my colleagues to not only confront our own ideas and solutions, but also to try and think about ways of how our codes could bbe improved. The pictures below show all the peer reviews done by me.

- Lab 2

Hi Lorenzo. I took a look at your code and these are the main things I noticed:

- You didn't consider the fact that we could have a maximum number of matches we can take with a single move.
- I like the idea of testing your best individual against different opponents at the end, but at this point it could have been better if your fitness function allowed you to make the various individuals play against the other opponents and not just the one using the optimal strategy.
- Considering that you are adopting the μ+λ strategy (thus you are not replacing the previous generation with the new one unless the latter's individuals have better fitness values), there's no need for the **best** variable to keep track of the current best individual since you are always sorting the list of all the individuals by their fitness value. All you have to do is looking at the last member of the list.
- The idea of plotting the best fitness value for every generation is great. It helps showing at which rate your algorithm is progressing.

In the end, I think you did a pretty good job and your results seem good.

Hi Paola. I took a look at your code at your code and this is what I observed:

- The code itself is pretty easy to understand. Only exception might be the mutation function that took me some extra effort, but still manageable.
- Having to modify the parameters when switching from first to second strategy or viceversa can be a bother. It would be a lot more comfortable and quick if those parameters could be set inside those 2 functions directly.
- There isn't really a reason to store the best individual and the best fitness in extra variables since these values are already present in the population list which is also sorted based on the fitness value at every generation. Simply checking the last element of the list would provide you the best individual and its fitness value.
- When creating the list of all the possible moves with the second strategy, what you're doing is just saying how many objects you can take at most from each row, meaning that those are not *ALL* the possible moves.

Despite these observasions, your comprehension of the topic is pretty obvious and the results you got with your algorithm are definitely good.

- Lab 9

Hi Lorenzo. I looked at your code and this is what I noticed:

- The different implementations of crossover and tournament selection show a deep understanding of how EA algorithms work. The implementation of mutation is fine as well but 50% chance for every single gene to mutate may be too high cause then the new individual would just end up being completely different from both parents and at this point you could just mutate a single individual without any recombination at all.
- The implementations of the island model is decent but also quite basic. You could have tried to add other features like migration to try to encrease diversity.
- You could have tried to increase the number of iterations to see how much of a difference it could have made in terms of best fitness score.

I hope my suggestions can be useful for you.

Hi Paola. I read your code and this is what I noticed:

- The way you implemented both uniform and 2 point crossover indicates a deep understanding of the topic. I also liked the fact that you provided a crossover function that does the one or the other based on a flag. Makes it a little easier to switch between one or the other.
- I like that the extinction function first looks if there are some repeated individuals rather than just removing individuals who don't have the highest fitness.
- You could have shown the graphic for the problem with a dimension higher than 1 just to see how the algorythm works in those other cases, but the fitness growth seems good in that instance of the problem.

In general, your knowledge of EA algorythms seems pretty good and your code also seems pretty versatile

8. Quixo

Now all that there's left to discuss is the final project. For this academic year, it was about creating an agent able to play Quixo against an opponent who plays randomly. The game is turn based and it's played on a board of dimension 5x5. On each turn a player selects a cell belonging to the perimeter of the board and which is also still not taken or taken by himself, removes that cell temporarily, slides the row or column of the cell in one of 4 possible directions and then puts the taken cell in the spot that's left empty. A player wins when he manages to align 5 of his own cells, whether it's by row, column or diagonal, without aligning 5 of the opponent's cells as well (if it happens, the player who made the move loses the game). This project was realized with my colleague Matassa Paola.

To create the agent, I resorted to Adversarial Search, which means that the player looks for the optimal move by going down a tree. More specifically, I used the MinMax algorithm. Since the branching factor for the Quixo game is very high, we had to adopt a few strategies to get around the problem.

```
def minimax(self, game:Game, depth, alpha, beta, isMaximizing):
    if depth == 0 or game.check_winner() != -1:
        return self.evaluate(game, self.player_id), None
    if isMaximizing:
        best_move = None
        v = -float('inf')
        valid_moves = get_all_valid_moves(game, self.player_id, self.with_simmetries) # mo
        for vm in valid_moves:
            next_game = deepcopy(game)
            next_game._Game__move(vm[0][0], vm[0][1], self.player_id)
            eval,_ = self.minimax(next_game, depth-1 , alpha, beta, False)
            if eval > v:
                v = eval
                best_move = vm[0]
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return v, best_move
    else:
        best_move = None
        v = float('inf')
        valid_moves = get_all_valid_moves(game, 1 - self.player_id, self.with_simmetries)
        for vm in valid_moves:
            next_game = deepcopy(game)
            next_game._Game__move(vm[0][0], vm[0][1], 1 - self.player_id)
            eval, _ = self.minimax(next_game, depth - 1, alpha, beta, True)
            if eval < v:
                v = eval
                best_move = vm[0]
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return v, best_move
```

First, we had to implement the alpha beta pruning to avoid expanding nodes that we knew wouldn't change the result. Then, we also set a maximum depth beyond which there's no further expansion.

```
def evaluate(self, game: Game, player_id) -> int:
    winner = game.check_winner()
    if winner == player_id:
        return 1
    if winner != -1:
        return -1
    my_count = count_max_aligned(game.get_board(), player_id)
    opp_count = count_max_aligned(game.get_board(), 1 - player_id)
    return (my_count - opp_count) / 5
    # return 0
```

If we get to the maximum depth without reaching a terminal state (win or loss), then the score of the node is obtained by computing the maximum number of aligned objects from both my player and the opponent and dividing their difference by 5 so that the score value would be between -1 and 1, corresponding to losing and winning state respectively.

```python
def check_simmetries(list_states, new_state, free_cells_new_state, one_cells_new_state):  #move => ((i, j), slide)

    for s in list_states:
        free_cells = np.count_nonzero(s[1]==-1)
        one_cells = np.count_nonzero(s[1]==1)

        if free_cells_new_state != free_cells or one_cells_new_state != one_cells or new_state[2][2] != s[1][2][2]:
            return False

        for k in range(1, 5): # k = 4 means 4 rotations so it's like not rotating it
            for flip in [1, -1]:  # 1 => not flipped, -1 => flipped
                key = k * flip
                int_state = np.flip(s[1], axis=1) if key<0 else s[1]# axis=1 is to flip horizontally
                int_state = np.rot90(int_state, k=abs(k)%4, axes=(1, 0))
                if np.array_equal(new_state, int_state): # you found the simmetry, now the fun starts
                    return True
    return False
```

The last thing we tried to reduce the branching factor was the exploitation of symmetries. When we try to expand a certain state, first it computes all the possible moves a player can do and all the states they would generate, but then it discards all the states that are symmetrical to one of the already discovered states. There are 8 possible symmetries that we tested, given by a combination of rotation (0°, 90°, 180°, 270°) and flip. This allows to substantially reduce the branching factor, especially at the beginning of the game, but apparently computing all the possible symmetries is very expensive and the player is actually slower when trying to exploit them, or it's at least true for the maximum depths that we tried (3 and 4).

```
TEST GAMES
100%|██████████| 20/20 [03:38<00:00, 10.91s/it]
wins: 100.00%, losses: 0.00%
```

max depth=3 and with symmetries

```
TEST GAMES
100%|██████████| 20/20 [02:08<00:00,  6.43s/it]
wins: 100.00%, losses: 0.00%
```

max depth=3 and without symmetries

As shown in the picture above, our player won all the games we made him play. Also, just to test our player further, we created another player class that allows a human player to play against the player using MinMax.

```python
class HumanPlayer(Player):
    def __init__(self) -> None:
        super().__init__()

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        print_board(game)
        print()
        while True:
            from_pos = input("Enter the position of the piece you want to move (x,y): ").split(',')
            if from_pos[0]=="ESCAPE" or from_pos[0]=="escape" or from_pos[0]=="\033":
                exit()
            if not from_pos[0].isdigit() or not from_pos[1].isdigit():
                print("Insert numbers, try again")
                continue
            else:
                from_pos = tuple(map(int, from_pos))
                if from_pos[0]>=0 and from_pos[0]<=4 and from_pos[1]>=0 and from_pos[1]<=4:
                    break
                else:
                    print("Numbers too big, try again")
        move = input("Enter the direction you want to move the piece (TOP, BOTTOM, LEFT, RIGHT): ")
        if move == "TOP" or move=="top":
            move = Move.TOP
        elif move == "BOTTOM" or move=="bottom":
            move = Move.BOTTOM
        elif move == "LEFT" or move=="left":
            move = Move.LEFT
        elif move == "RIGHT" or move=="right":
            move = Move.RIGHT
        else:
            print("slide not exist, try again")
        print("YOU: ", from_pos, move)
        return (from_pos[1], from_pos[0]), move
```

After a few matches, we noticed that our player was usually able to identify if the current state of the board was not good for him and he was therefore able to avoid performing moves that could have brought us into a winning situation.