Report: Computational Intelligence Laboratories

Lorenzo Fezza s312689

January 31, 2024

1 Introduction

This report contains a detailed explanation of the activities carried out in the laboratories addressed during the 2023/2024 academic year for the Computational intelligence (PoliTo) course. Let's begin.

2 Laboratories

Lab 1. Set Covering Problem

Submitted: 25/10/2023

Set Covering is an optimization problem which, given a finite set of elements, requires to find the minimum subset of elements whose union covers the entire initial set. This problem can be solved with different optimization strategies, such as the path search, the main topic addressed during the first period of lessons. In particular there are different path search algorithms, which can follow Informed or Uninformed strategies, which differs from the fact that, in Uninformed strategies, the algorithm has no additional information about the search space or the destination's location. Informed strategies instead, allow to find the solution in a more efficient way providing good heuristics which can guide the solution search.
The algorithm required for the implementation of this laboratory is A* algorithm. It is an informed strategy which consist in computing the path with minimum cost by expanding only the nodes that bring you closer to the solution. It is complete and optimally efficient because it expands the minimum number of nodes, starting from the best one. In order to make decisions on what nodes to expand, an evaluation (fitness) function of the

actual state is needed. In the A* algorithm, the fitness function is formulated as $f(n) = g(n) + h(n)$ where n is the node in the tree, g is used to evaluate the node and h is a monotonic function used to estimate the cost of that node (so it is a heuristic). The A* algorithm works only under reasonable assumptions: h(*) is admissible (if it never overestimates the cost to reach the destination node) and the branching factor is finite. In the set covering problem the search space is a tree and choosing h(n) as the number of sets taken for the solution, it can be considered an admissible heuristic, which guarantees optimal solutions. The goal is to minimize the number of sets so, it's reasonable to choose it as a cost function. The more sets taken, the higher the cost.

The following code represents the fitness function f:

```python
def f(new_state):
    return distance(new_state) + len(new_state.taken)
```

where:

- `distance(new_state)` computes the union of the taken sets by computing the `or` operation element by element and returns the number of uncovered elements (`PROBLEM_SIZE - covered elements`)
- `len(new_state.taken)` is the number of sets taken

The tree is explored in an iterative way with the following cycle, by putting a new set (represented by the action variable) one by one in the `new_state` variable (removing it from the not taken list). The value of the new state is then put in the frontier (a priority queue) if it comes closer to the solution. The evaluation is performed by the f function presented above.

```python
_, current_state = frontier.get()
while not goal_check(current_state):
    for action in current_state[1]:
        new_state = State(
            current_state.taken ^ {action},
            current_state.not_taken ^ {action},
        )
        if f(new_state) < d_curr:
            frontier.put((f(new_state), new_state))
    d_curr, current_state = frontier.get()

print(
    f"{len(current_state.taken) tiles)"
)
```

The tree is expanded starting from the node with less distance from the full coverage state and has less number of sets.

Halloween Challenge.
Set Covering Problem with Hill Climber solution

Submitted: 31/10/2023

The set covering problem can be solved with many other different strategies, for example with the use of the Hill Climbing, a single-state method. It considers a single state at a time and it progressively improves the initial state until a local optima or the optimal result is reached.
The Hill Climbing method consists imaging the objective function as a hill. An agent is able to climb this hill tweaking the actual state finding a new one which is able to improve the actual state, getting better results in the objective function.
The goal of this challenge was to find the best solution with the fewest calls to the fitness functions and the Hill Climb can address this problem by tweaking instead of walking along the tree calling the fitness function each time.

The `tweak(state, ffid)` method is implemented as below:

```python
def tweak(state, ffid):
    new_state = copy(state)
    rows_with_true_at_first_false_index = np.where(np.
       array(x[:, [ffid]].toarray()))[0]
    rows_not_in_new_state = np.setdiff1d(
       rows_with_true_at_first_false_index, np.where(
       new_state))

    if len(rows_not_in_new_state) > 0:
        random_row_index = np.random.choice(
           rows_not_in_new_state)
        # select a random row with first false index (
           ffid) not in the actual state
        new_state[random_row_index] = not new_state[
           random_row_index]
    else:
        # randomly remove an element from the actual
           state
        random_row_index = np.random.choice(np.where(
           new_state))
        new_state[random_row_index] = not new_state[
           random_row_index]
    return new_state
```

A new set is randomly chosen from a set of rows which covers the first uncovered element at index ffid. It is not necessary to compute the line 4 of this code because it's clear that the rows selected in line 3 are not present in the current state, if that rows have that index equal to false (the union of the current state rows does not cover that index).

After the new row is selected, the fitness function (used to evaluate the quality of the solution) is called on the new state and on the current state and the values are compared. If the quality increases, the new state becomes the actual state, the agent moves upper to the hill and the new first false index is calculated again, in order to pass it to the new tweak call. The algorithm stops when the full set is covered.

```python
# the initial state is initialized with 0 taken sets, but it can be modified to choose some random sets, which later on can be evaluated as useful or not
current_state = [choice([False, False, False, False, False, False]) for _ in range(NUM_SETS)]

cnt = 0
diff = 0
first_false_index = 0
fit_cnt = 0

for step in range(10_000):
    new_state = tweak(current_state, first_false_index)
    fn = fitness(new_state)
    fc = fitness(current_state)
    fit_cnt += 1
    if fn >= fc:
        result_of_reduce = reduce(
            np.logical_or,
            [np.array(x.getrow(i).toarray()[0]) for i, t in enumerate(new_state) if t],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        )

        first_false_index = None
        for i, value in enumerate(result_of_reduce):
            if not value:
                first_false_index = i
                break

        if first_false_index is None:
            print("Tutti i valori sono veri.")
            current_state = new_state
            break

        cnt += 1
        current_state = new_state
        #print(fitness(current_state))

print(f"Steps: {cnt}")
print(f"Fitness calls: {fit_cnt}")
```

Unfortunately the implementation of this algorithm does not give the optimal result, but reaches a local optimum, in fact for 5000 sets with 5000 elements with a density of covered elemnts equal to 0.3, it takes about 20/21 sets as solution, but it calls the fitness function only 19/20 times (one time less than the number of selected sets).

Lab 2. Nim Game with Evolutionary Strategy

Submitted: 17/11/2023

For this laboratory, it was necessary to implement an evolutionary strategy to play the nim game with a rule based system. In particular the implemented algorithm involves a phase of training and than the parameters learned and optimized are used to test the performances obtained.

The training phase aims to determine optimal parameters for the algorithm. The process involves an evolutionary strategy that uses a population of agents that plays the nim game to determine the best parameters. Before the implementation of the Evolutionary Strategy, the phenotype of the agent is defined. The phenotype is defined by the following properties:

- weights: is a list of 3 values, which represents the probabilities to choose a rule with respect to another one;
- fitness: is a value which represents the percentage of victories using my strategy with respect to the optimal one;
- sigma: is the standard deviation of the gaussian distribution used to mutate the weights.

The first method implemented is the `fitness` function which consists of a function that, given the number of rows (`n_rows`) used to build the nim game and the parameters to optimize `(vals)`, plays the game for `(trainings)` times using the given optimal strategy (player = 0) and `my_strategy` (player = 1) implemented later. The percentage of victories is returned and used to determine the fitness value in the phenotype of the evolutionary strategy.

```python
strategy = (optimal, my_strategy)

def fitness( weights) -> float:
    """The fitness function"""
    win_cnt = 0 # number of wins for the candidate
    for _ in range(trainings):
        nim =  Nim(N_ROWS)
        player = 0
        while nim:
            if player == 0:
                ply = strategy[player](nim)
            else:
                ply = strategy[player](nim, weights)
            nim.nimming(ply)
            player = 1 - player
        if player == 1:
            win_cnt += 1
    return win_cnt / trainings * 100
```

The second method implemented is the `my_strategy()` function, which takes two parameters as input: the `state` (nim table) and the `weights`. The `state` is used to determine all the decisions that the player can take. The choice is given by the `weights` list. Thanks to the `numpy.rand.choice()` function, given that three rules are defined and the `weights` list represents the probabilities to take an action, a rule is chosen. At the beginning, different rules were used in the strategy, but only three of them are significant and used in the code:

1. Take all the items from the row with more items.
2. Take one item from the row with fewer items.
3. Take a random choice.

```python
def my_strategy(state: Nim, weights) -> Nimply:
    spicy_moves = [Nimply(r, o) for r, c in enumerate(
       state.rows) for o in range(1, c + 1) if c > 0]
    # for each value of weights, assign a decision to
       take
    #       for example:
    #       0: take all the elements from the row with the most objects
    #       1: take one item from the row with less objects
    #       else: take a random row

    rule = np.random.choice(len(weights), p=weights)
    if rule == 0:
        return max(spicy_moves, key=lambda move: move.num_objects)
    if rule == 1:
        return min(spicy_moves, key=lambda move: move.num_objects)
    return random.choice(spicy_moves)
```

The third method implemented is the `ES()` function. The evolutionary strategy $(\mu + \lambda)$ follows these three steps:

1. Initialization: The function initializes `mu` individuals in the population, giving the same probability for each rule. Thus, each element of `weights` is the same. The `fitness` is calculated using these `weights`. Then, (`weights`, `fitness`, `sigma`) are put in the population as new individuals.

2. Child generation or mutation: In this second step, the population is mutated using the `mutation()` function, which takes as input the initial population and generates `lamb` children. The new individuals are concatenated to the initial population and put in the `offspring` variable (new population composed of `mu + lamb` individuals), and the individual with the best fitness value is selected and compared to the best solution. If the new individual is better than the best one, it is replaced.

3. New population selection: The best `mu` individuals are selected and put in the new population as survivors.

```
mu = 5  # number of population
lamb = 5 * mu  # size of children
sigma = 1  # mutation strength
trainings = 20  # number of training games per candidate
N_ROWS = 5  # number of rows in the game

```

```python
def ES():
    P = []
    history = list()

    #initialize population
    for _ in range(mu):
        weights = np.random.dirichlet(np.ones(3))    # because I'm using 3 rules
        fit = fitness(weights)
        # print(fit)
        P.append((weights, fit, sigma))

    Best = None
    for step in tqdm(range(300)):
        offspring = P + mutation(P) # (mu + lambda) strategy
        offspring = sorted(offspring, key=lambda i : i[1])
        if Best is None or offspring[-1][1] > Best[1]:
            Best = offspring[-1]
            print(f"Best weights: {Best[0]}, fitness: {Best[1]}, sigma: {Best[2]}")
            history.append((step, Best[1]))
        P = offspring[-mu :]
    history = np.array(history)
    plt.figure(figsize=(14, 4))
    plt.plot(history[:, 0], history[:, 1], marker=".")
    return Best
```

The last method implemented is the `mutation()` function. Initially empty, it was implemented to generate new individuals starting from the initial population. The function takes as input the initial population and generates `lamb/mu` children for each parent. Since the starting population is composed of `mu` parents, the new individuals generated will be `lamb` in total. The mutation is done by introducing random noise in the weights, randomly varying the weights to find a combination that maximizes the `fitness` value. The `sigma` variable is used to vary the amount of mutation applied to the weights during the evolution of the population, but it's constant over time.

```python
def mutation(P):
    """Mutate the population"""
    offspring = []
    for i in P:
        for _ in range(lamb//mu):
            sigma = np.abs(np.random.normal(loc=i[2], scale=0.2))
            mutated_weights = np.abs(np.random.normal(loc=i[0], scale=sigma))
            mutated_weights /= sum(mutated_weights) # normalizing weights
            mutated_fitness = fitness(mutated_weights)
            offspring.append((mutated_weights, mutated_fitness, sigma))
    return offspring
```

The following image shows how the best fitness value changes during the evolution of the population:

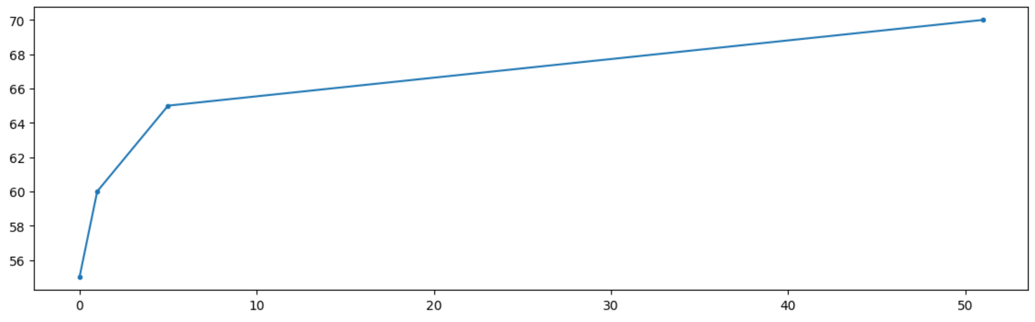


Figure 1: Fitness value evolution over iterations

A game can be played as following:

```
strategy = (optimal, adaptive)

nim = Nim(N_ROWS)

logging.info(f"init : {nim}")
player = 0
while nim:
    ply = strategy[player](nim)
    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!")
```

where `adaptive` is initialized as following:

```
genome = ES()
```

```python
print(f"Best weights: {genome[0]}, fitness: {genome[1]},
    sigma: {genome[2]}")

def adaptive(state: Nim) -> Nimply:
    """A strategy that can adapt its parameters"""
    return my_strategy(state, genome[0])
```

After the training phase is complete, the precomputed optimal weights are used to play the `adaptive` strategy against `optimal`, `pure_random`, and `gabriele` strategies.

The evolutionary strategy generates promising results playing against the optimal strategy during the learning phase. However, these results are good against Gabriele, but not as good against optimal and pure random players in the test phase.

The results over 100 test games are shown in the following table:

Strategy	**%w**
Optimal	34
Pure_Random	61
Gabriele	89

Peer review Lab 2.

Submitted: 22/11/2023

Lab 9. Abstract Problem with Evolutionary Algorithm solution

Submitted: 03/12/2023

This laboratory requires the implementation of a local-search algorithm able to solve the Problem instances 1, 2, 5 and 10 (a black box task) on a 1000-loci genomes using a minimum number of fitness calls.

A local search, a heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed.

The local-search implemented in this laboratory is a genetic algorithm.

The first idea to solve this black box task was to think the fitness function as an eye to look for what pieces of the bitstring are important and which are not and therefore to preserve the important pieces and throw away the useless ones. Unfortunately this last idea is not covered in the following implementation but the most of the code is related to this.

Another important consideration to solve this task is the promotion of diversity. The evaluation of different individuals in the population can help to get closer to the solution when the problem is not known.

Diversity is evaluated using a distance metric, which indicates how far the individual is from a subset of the population to the whole population and therefore it is a property of the population.

Three levels of diversity can be exploited:

- phenotype
- genotype
- fitness

For the implementation of this laboratory, different values, such as the population size μ, problems *problems*, the size of the bitstring l and others are parametrized in order to simplify the evaluation of the performances of the algorithm.

Methods :

`genetic_algorithm() :`

The first method implemented is the skeleton of the `genetic_algorithm()` function. At the beginning this function only had 1 parameter, namely the fitness function. After that, other parameters were introduced to evaluate different mutations, crossovers or parent selections in a simpler way.

This function executes the following steps:

- Initialization of the population
- Evaluation (fitness) of each individual
- Loop until the time is over or the stopping criterion (100% returned by fitness) is met:
  - Update of the best individual
  - Initialize a new population (initially empty)
  - Loop for half of the population size:

    * Selection of two parents
    * Crossover of the parents, generating two children (offspring)
    * Mutation of the offspring
    * Add the offspring to the new population
  - Evaluation (fitness) of the new population
  - Update of the population

Than the best individual fitness value, with the count of fitness and the iteration where the best individual is found is returned. This results can be significant in order to reduce the external loops.

Code:

```
def genetic_algorithm(fitness, selection, crossover):
    global saved_matrices
    saved_matrices = []
    Best = None
    # 1. Initialize population
    population = init_population()
    population = evaluate_population(population, fitness
       )
    # 2. Repeat
    found = -1
    x = -1
    for i in range(iterations):
        for p in population:
            if Best is None or p[1] > Best[1]:
                if i!=0:
                    print_matrix()
                print(f"Best: {p[0][0:5]}...\tfitness:{p
                   [1]}\tcalls:{fitness._calls}")
                Best = p
                found = fitness._calls
                x = i

        if Best is not None and Best[1]==1:
            break

        q = list()
        for _ in range(mu//2):
            # 2.1 Select parents
            parent_a, parent_b = selection(population)
            # 2.2 Crossover
```

```
                child_a, child_b = crossover(copy(parent_a),
                    copy(parent_b))

                # 2.3 Mutate
                mutated_a = mutate(child_a)
                mutated_b = mutate(child_b)
                q.append(mutated_a)
                q.append(mutated_b)

            population = evaluate_population(q, fitness)

    # 4. Return best individual
    plot()
    return Best, found, x
```

Some global variables are used to save data used to evaluate the process and results and the method used inside were implemented in a second time, than modified more and more times.

`init_population() :`

Initializes randomly the population, composed by µ individuals, each one with a bitstring of l bits and a fitness value of 0 at the beginning.

```
def init_population():
    return np.array([(np.array(choices([0, 1], k=l)),
       0.0) for _ in range(mu)], dtype=object)
```

`evaluate_population() :`

Evaluates the population using the fitness function. The fitness value is calculated for each individual and the tuple is updated.

```
def evaluate_population(population, fitness):
    return np.array([(individual[0], fitness(individual
       [0])) for individual in population], dtype=object
       )
```

`select_with_replacement() :`

This method initially took two random individuals from the population but now selects elements based on their mutual diversity. This diversity is computed for each individual and all the other individuals. The diversity is then weighted with the fitness value and evaluated, and the two individuals with the highest value are selected.

Remember: the diversity can be evaluated considering

- Genotype: bit-string of l bits
- Phenotype: bit-string of l bits
- Fitness: function fitness returned by `lab9_lib.make_problem(problem)`

```python
def select_with_replacement(population):
    global diff_matrix
    diff_matrix = np.zeros((mu, mu))
    for i1, p1 in enumerate(population):
        for i2, p2 in enumerate(population):
            # simmetric matrix
            diff_matrix[i1][i2] = compute_diversity(p1
                [0], p2[0])
    return random.choice(population), random.choice(
        population)
```

`compute_diversity()` :

In order to evaluate diversity, a XOR operation is performed between two individuals. The result of the XOR is then used to calculate the percentage of different bits with respect to the total number of bits ($l = 1000$). It represents genotype diversity.

```python
def compute_diversity(e1, e2):
    xor_res = e1 ^ e2
    return np.sum(xor_res) / l
```

Considerations:

During the `select_with_replacement()` implementation, the diversity matrix was printed in order to understand if the diversity was correctly computed and if the fitness value could influence the selection. Using the colorama library it was possible to highlight the maximum diversity value of the matrix and the value selected weighting with the fitness value. From this test it was possible to understand that usually the selected values are also the most different ones.

Later on, the diversity matrix of the population that generated the children that gave rise to the best (that matrix which is not weighted with the fitness value) was plotted in order to understand how the diversiry is distributed during the evolution of the algorithm. Those graphs are very nice and some of them are shown below (Fig. 2, 3, 4).

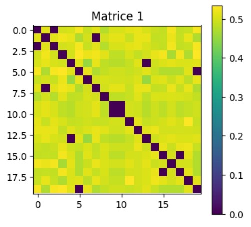


Figure 2: Diversity matrix which provided the best population for solving Problem 2. with std crossover and std select with replacement

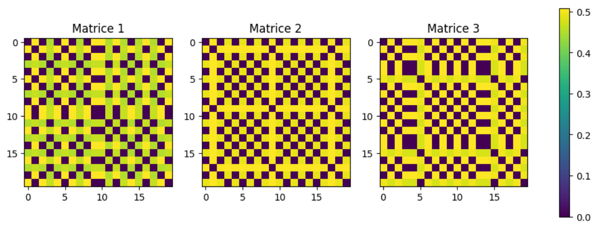


Figure 3: Diversity matrices which provided the best population for solving Problem 5. with cyclic shift crossover and selection based on diversity (evolution of div_matr)

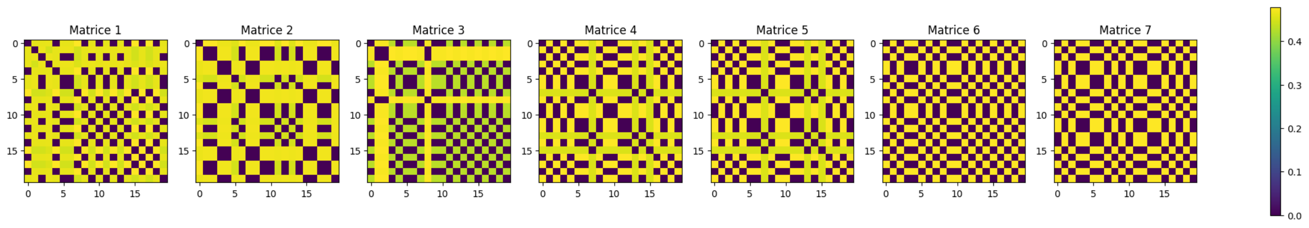


Figure 4: Diversity matrices which provided the best population for solving Problem 1. with cyclic shift crossover and selection based on diversity (evolution of div_matr)

`diversity_selection() :`

It is used to select the most different individuals, weighting the difference with the fitness value. Also in this case the `diff_matrix` is modified in order to evaluate the diversity convergence over the generations. A `study_diversity_selection()`

method was used at the beginning to study the evolution.

```
def study_diversity_selection(population):
    global fit1, fit2, max1, max2, diff_matrix
    # find the individuals with
    # the highest fitness and the highest diversity
    div_matr = np.zeros((mu, mu))
    diff_matrix = np.zeros((mu, mu))
    for i1, p1 in enumerate(population):
        for i2, p2 in enumerate(population):
            if i1 != i2:
                #the matrix is not symmetric since is
                   added only for p2[1]
                div =compute_diversity(p1[0], p2[0])
                div_matr[i1][i2] = ( div + p1[1] + p2
                   [1]) / 3
                diff_matrix[i1][i2] = div

    # find index of the individuals with highest fitness
        and highest diversity
    i1, i2 = np.unravel_index(np.argmax(div_matr),
       div_matr.shape)
    fit1, fit2 = i1, i2
    max1, max2 = np.unravel_index(np.argmax(diff_matrix)
       , div_matr.shape)

    return population[i1], population[i2]

def diversity_selection(population):
    global fit1, fit2, max1, max2, diff_matrix
    diff_matrix = np.zeros((mu, mu))
    max_wdiv = 0
    max_div = 0
    i1, i2 = None, None
    for i, p1 in enumerate(population):
        for j, p2 in enumerate(population):
            if i < j:
                div_val = compute_diversity(p1[0], p2
                   [0])
                if div_val > max_div:
                    max_div = div_val
                    max1, max2 = i, j
                wdiv = ( div_val + p1[1] + p2[1]) / 3
                diff_matrix[i][j], diff_matrix[j][i]  =
                   div_val, div_val
                if wdiv > max_wdiv:
                    max_wdiv = wdiv
                    fit1, fit2 = i, j
                    i1, i2 = p1, p2
return i1, i2
```

`mutation()` :

This method takes as input an individual and a probability of mutation (set by default to ‘0.5‘) and computes a bit-flip mutation, returning a new mutated individual. The mutation is performed on each bit of the bitstring with the given probability in a for loop. If the random number generated is less than the probability, the bit is flipped.

```
def mutate(individual, mutation_prob=0.5):
    # bit flip mutation for now
    v = individual[0]
    for i in range(l):
        if mutation_prob >= random.random():
            v[i] = 1 - v[i]
    return individual
```

Two methods of crossover are implemented and their performance is evaluated in order to understand which one is better:

`std_crossover()` :

This method performs a two-point crossover between two individuals. The two points are randomly generated and the two sections of the bit-string are swapped between the two points.

```
def std_crossover(parent1, parent2):
    # a two (rand) point crossover now
    v = parent1[0]
    w = parent2[0]
    c = random.randint(0, l)
    d = random.randint(0, l)
    if c > d:
        c, d = d, c
    if c!=d:
        v[c:d], w[c:d] = w[c:d], v[c:d]
    return (v, 0.0), (w, 0.0)
```

`crossover_cyclic_shift()` :

This method performs a two-point crossover between two individuals but the two sections of swapping are shifted and the bit-string is treated as a circular array.

```python
def crossover_cyclic_shift(parent1, parent2):
    # circular translated swapping
    v = parent1[0]
    w = parent2[0]
    c = random.randint(0, l)
    d = random.randint(0, l)
    s = random.randint(0, l)
    if c < d:
        for i in range(c, d):
            v[i%l], w[(i+s)%l] = w[(i+s)%l], v[i%l]
    else:
        for i in range(c, d+l):
            v[i%l], w[(i+s)%l] = w[(i+s)%l], v[i%l]
    return (v, 0.0), (w, 0.0)
```

The cyclic shift crossover appears to work better than the standard crossover by increasing the fitness percentage by a few units.

Island Model :

After the implementation of the genetic algorithms with variants of parent selection and crossover, a non-parallelized island model is implemented, in order to observe how different populations can evolve, passing promising individuals among themselves. In the implemented model, an individual is promising if it has a high fitness value. Since this promising value is passed from one population to another it is likely that it is also different from other individuals in another population. Clearly, if the population is not very large, convergence to very similar individuals is rapid.

In particular this model iterates `iteration` times and, for each iteration, each island computes a cycle of `evolve()` method, in which two parents are selected thanks to the selection methods implemented above, than crossover and mutation are computed and the new population is evaluated in order to select the best individual to pass to another population.

```python
l = 1000
problems = [1, 2, 5, 10]
half_pop_size = 10
mu = 2 * half_pop_size
```

```python
iterations = 100

def i_diversity_selection(population):
  # find the individuals with
  # the highest fitness and the highest diversity
  max_div = 0
  i1, i2 = None, None
  for i, p1 in enumerate(population):
      for j, p2 in enumerate(population):
          if i < j:
            div = (compute_diversity(p1[0], p2[0]) + p1[1] + p2[1]) / 3
            if div > max_div:
              max_div = div
              i1, i2 = p1, p2
  return i1, i2

def evolve(population, fitness):
  Best = None
  q = list()
  for _ in range(mu//2):
      # Select parents
      parent_a, parent_b = i_diversity_selection(population)
      # Crossover
      child_a, child_b = crossover_cyclic_shift(copy(parent_a), copy(parent_b))
      # Mutate
      mutated_a = mutate(child_a)
      mutated_b = mutate(child_b)
      q.append(mutated_a)
      q.append(mutated_b)
  population = evaluate_population(q, fitness)
  idx_p = None
  for i, p in enumerate(population):
    if Best is None or p[1] > Best[1]:
      Best = p
      idx_p = i
  return Best, idx_p

def island_method(fitness, n_islands=2):
  populations = []
  Best = None
  tmp_best = None

  for _ in range(n_islands):
    populations.append(init_population())

  for _ in range(iterations):
    for i in range(n_islands):
      for j in range(n_islands):
        if i != j:
          i1, idx_1 = evolve(populations[i], fitness)
          i2, idx_2 = evolve(populations[j], fitness)
          populations[i][idx_1], populations[j][idx_2] = i2, i1  # exchange of the individual
          tmp_best  = max([i1, i2], key=lambda i: i[1])
          if Best is None or tmp_best[1] > Best[1]:
            Best = tmp_best
  return Best

for p in problems:
  fitness = lab9_lib.make_problem(p)
  b = island_method(fitness)
  print(f"Problem {p}:{fitness.calls}, best:{b[1]:.2%}")
```

Results :

The results are evaluated using the percentage of fitness reached. Unfortunately, the professor implementation, which generates 10 random bitstrings, calling only 10 times the fitness function, generates solutions slightly lower than the solutions obtained using the genetic algorithm which calls the fitness many more times.

Statistics :

Professor results (random search):

Problem	Fitness	Fit calls
1	53.90%	10
2	48.60%	10
5	19.50%	10
10	5.29%	10

Cyclic shift crossover using selection based on diversity:

Problem	Fitness	Fit calls
1	54.40%	2020
2	52.60%	2020
5	22.34%	2020
10	17.11%	2020

Standard crossover (two pt) using selection with replacement:

Problem	Fitness	Fit calls
1	54.00%	2020
2	52.40%	2020
5	20.62%	2020
10	16.20%	2020

Island model (using Cyclic shift crossover using selection based on diversity because of the best performances):

Problem	Fitness	Fit calls
1	54.40%	8000
2	52.00%	8000
5	31.24%	8000
10	25.43%	8000

Lab 10. Tic Tac Toe Player with Q-Learning

Submitted: 25/12/2023

This laboratory there is the implementation of a tic-tac-toe player using reinforcement learning, a type of machine learning where an agent learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties, allowing it to learn optimal strategies through trial and error and his goal is to maximize cumulative rewards over time by discovering effective actions in different situations. In particular the player is trained using the Q-Learning algorithm.

Let's start from the beginning.

The starting professor's code is not modified and the baseline of this laboratory is represented by the magic square (explained below). For the implementation of a reinforcement learning algorithm, it is necessary to define the following elements :

- **Environment :** the environment in this case is the tic-tac-toe game, represented by a board, filled with Xs and Os. The board can be easily printed with the provided method **print_board()**, which takes as input the actual state of the board and prints it putting dots for white spaces.

- **States :** the states are the possible configurations of the environment. In this case the states are the possible configurations of the tic-tac-toe game, represented by the provided code `State = namedtuple('State', ['x', 'o'])` where x and o are lists of the actions chosen by the two players.

- **Actions :** the possible actions that a player can perform on the environment are represented by `MAGIC = [2, 7, 6, 9, 5, 1, 4, 3, 8]`. Each number corresponds to a position on the board. This list is used to evaluate the winner of the game by using the sum method. The player that has a sum of 3 elements equal to 15 wins.

$$\begin{bmatrix} 2 & 7 & 6 \\ 9 & 5 & 1 \\ 4 & 3 & 8 \end{bmatrix}$$

- **Reward:** The reward is the value that the agent receives for each action. The reward is implemented in the `get_reward()` method. Different versions of this method have been implemented to evaluate the performances of the agent learning.

  - The first version is the simplest one (`state_value()`), in which the agent receives a reward of 1 if it wins, -1 if it loses, and 0 if it draws.
  - The second version is an evolution of the first, implemented in `get_reward_0()`. It assigns +5 if the agent wins, -10 if it loses, and 0 if it draws.

```python
def get_reward_0(state, done):
    if win(state.o):
        return -10, True
    elif win(state.x):
        return 5, True
    else:
        return 0, done
```

  - The third version `get_reward_1()` changes the values of the previous `get_reward` with +50 if it wins and -100 if it loses. In addition, it adds a penalty of -3 for each action of x on its turn that did not block the opponent from winning because the opponent will win in the next move. If the action of x is blocking the winning of o, reward += 1. If x sets 3 elements at the corner (a configuration that facilitates the win), reward += 1. If x wins, reward += 50.

```python
target_set = {2, 4, 6, 8}

def get_reward_1(state, done, available_actions, turn=1):
    if win(state.o):
      return -100, True

    reward = 0
    count_combinations = 0

    if turn == 1: # if turn of x
      # if o can win with one action from the remaining and x is not blocking it => reward -= 3
      # o can perform it the next turn !
      elements = list(state.o)
      for act in available_actions:
        elements.append(act+1)
        if any(sum(c) == 15 for c in combinations(elements, 3)):
          reward -= 3
        elements.remove(act+1)

    # if the action of x is blocking the winning of o => reward += 1
    for x in state.x:
      elements = list(state.o)
      elements.append(x)
      count_combinations += sum(1 for c in combinations(elements, 3) if sum(c) == 15)
      elements.remove(x)
    reward += count_combinations
    # if x sets 3 elements at the corner (config that facilitates the win)
    cmn = state.x & target_set                  # reward += sum(1 for c in combinations(state.x, 3) if c in target_vector)
    if len(cmn) >= 3:
      reward += len(cmn) - 2

    if win(state.x):
      '''print("win: x")'''
      reward += 50
      return reward, True
    else:
      return reward, done
```

– The fourth version `get_reward_2()` is similar to the previous one, but the order of the operations is performed in a different way to evaluate the loss and victory at the beginning. If o wins, the reward returned is -100; if x wins, the reward returned is +50. Then, all the following things are evaluated:

  * If during the turn of x, there still remains actions that can make o win in a single move, reward returned is -100.
  * Reward += 10 if x blocks the winning of o.
  * Reward += 5 if x sets 3 elements at the corner.

```
def get_reward_2(state, done, available_actions,
    turn=1):
  if win(state.o):
    return -100, True

  if win(state.x):
    return 50, True

  if turn == 1: # if turn of x
    # if o can win with one action from the
       remaining and x is not blocking it =>
       reward = 100
    # o can perform it the next turn !
    elements = list(state.o)
    for act in available_actions:
      elements.append(act+1)
      if any(sum(c) == 15 for c in combinations(
         elements, 3)):
        return -100, done
      elements.remove(act+1)

  reward = 0

  # reward += 10 for each action of x that
     blocks the winning of o
  elements = list(state.o)
  for x in state.x:
```

```python
    elements.append(x)
    reward += 10 * sum(1 for c in combinations(elements, 3) if sum(c) == 15)
    elements.remove(x)

  # if x sets 3 elements at the corner (config that facilitates the win)
    cmn = state.x & target_set                # reward += sum(1 for c in combinations(state.x, 3) if c in target_vector)
    if len(cmn) >= 3:
      reward += 5*(len(cmn) - 2)

  return reward, done
```

Training Phase :

The training phase consists of a loop of `episodes` iterations, where the agent constantly updates his table to find the best strategy for the maximum reward. Initially, the training is performed using a `random_player()` as an opponent, but later, a more expert player, `my_player()`, is used to train the agent for better strategies. The expert player performs random actions, wins when possible, and blocks the opponent when it can win.

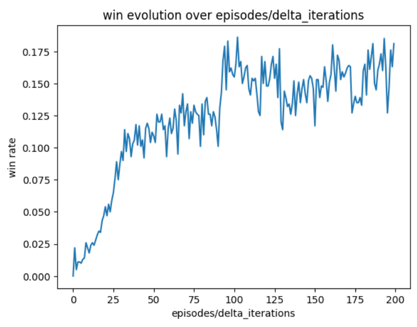
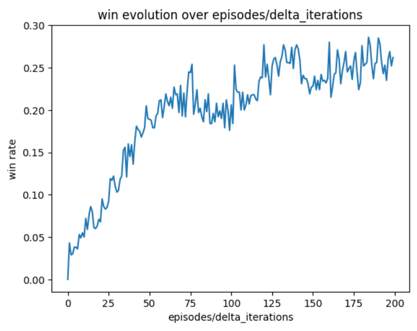
The agent begins learning from a random state and gradually chooses the best action based on the table. To balance exploration and exploitation, the GLIE strategy (Greedy in the Limit with Infinite Exploration) is adopted. Initially, the agent lacks knowledge and chooses random actions more often for given states to explore. As the training progresses, the agent updates the table, assigning values to each state-action pair, gradually increasing the exploitation phase.

To implement this, at each episode, epsilon (ϵ) is gradually decreased using the formula

$$\epsilon_k = \frac{b}{b+k}$$

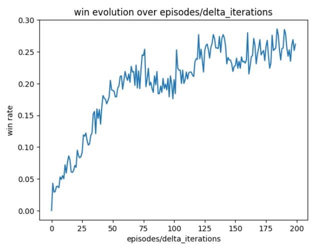
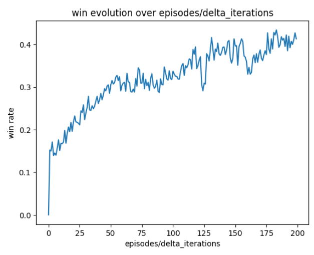
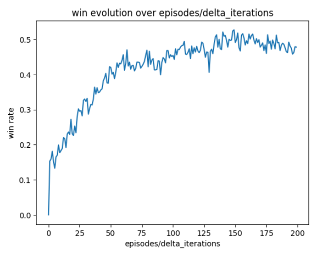
where k is the episode where ϵ will be used, and b is a constant chosen to make the value reach 0.1 at the end of the training phase.

The benefits of this approach are displayed in the following images, where the agent is trained for 200,000 episodes with a random beginner using `get_reward_0()`, `my_player()`.

Fixed epsilon	**GLE**
	

```python
for i in tqdm(range(episodes)):
    state = State(set(), set())
    done = False
    available_actions = list(range(9))
    turn = np.random.choice([0,1])
    epsilon = b / (b + i)
    while not done:
        flag = 0
        if turn == 1: # if turn of x
          # epsilon greedy player
          action = a.get_action(state, available_actions, epsilon)
          # fixed eps => don't pass epsilon parameter to get_action because it's fixed by default
          # action = a.get_action(state, available_actions)

        else:  # if turn of o
          action, flag = my_player(state, available_actions, flag)

        next_state, done = perform_action(action, available_actions, deepcopy(state), turn)

        reward, done = get_reward_0(next_state, done)

        a.update(state, action, reward, next_state, done)
        state = next_state
```

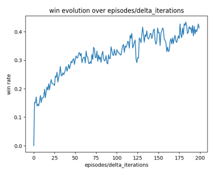
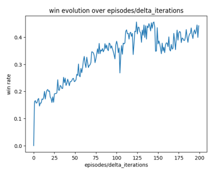
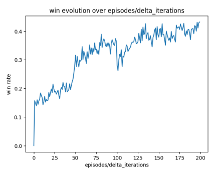
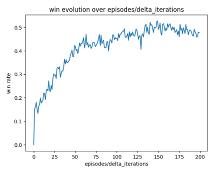
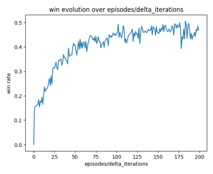
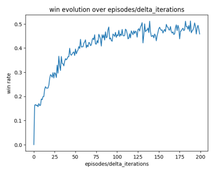
Different training loops can be visualized by plotting wins over the episodes. Considering an episode interval of `d_iter = 1000`, the percentage of wins in that interval is calculated and plotted. The results obtained are as follows:

get_reward_0	**get_reward_1**	**get_reward_2**
		

The training is performed using:

- GLIE approach
- The first turn is random
- `episodes`
- `my_player()`
- `get_reward_0/1/2()`

Performing the same training multiple times does not always provide the same results due to the random exploration phase, which differs in each training. In the same number of episodes, the agent may learn more or less important things, making it more or less expert in the game.

Training	**First Training**	**Second Training**	**Third Training**
`get_reward_1`			
`get_reward_2`			

Saving and Loading Phase :

In order to evaluate the performance of the agent and use the knowledge acquired during the training phase for many tests, the table is saved in a json file by the agent using the `save_q_table()` method. The table can be loaded using the `load_q_table()` method and used to test the knowledge learnt.

```
agent.save_q_table("q_table.json")
agent.load_q_table("q_table.json")
```

Testing Phase :

For this phase, the agent plays 1000 games against a random player and against the player from which it learns. Two different types of tests have been performed:

- **Random Player Test**: The agent plays 1000 games against a random player.
- **My Player Test**: The agent plays 1000 games against the player from which it learns.

The results obtained are reported in Table 1.

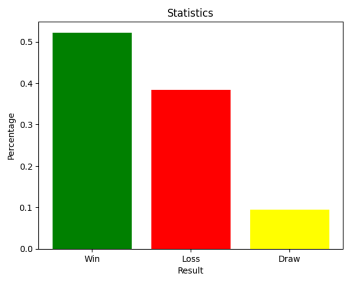
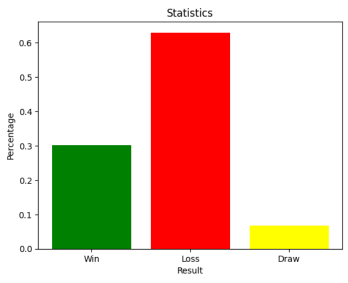
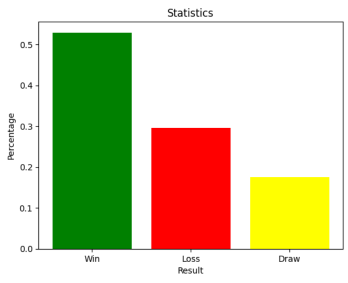
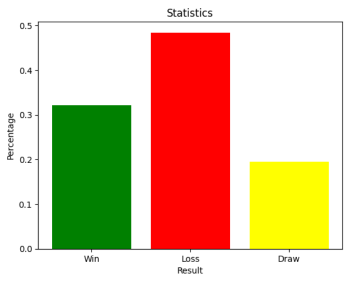
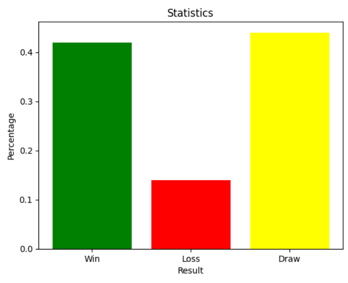
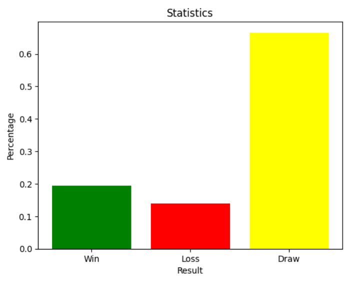
Test	Random Player Test	My Player Test
get_reward_0		
get_reward_1		
get_reward_2		

Table 1: Different training results

Exploiting Symmetries :

In order to exploit the symmetries, the update function is modified to update the table for all symmetric states. A rotation of 90°, 180°, and 270° is performed on the board using a support dictionary `SYMM`, which is used in a cycle to map the state to the corresponding rotated one. The rotation is performed only if it makes sense, i.e., if the only action performed at state s is 5 (the center), the rotation is not performed. If there is more than one action performed, the update is propagated to the other symmetric states-actions. The results obtained are shown below in Table 2. and compared with training without exploiting symmetries.

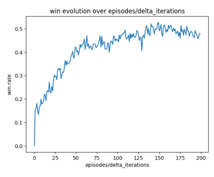
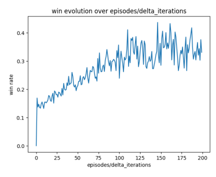
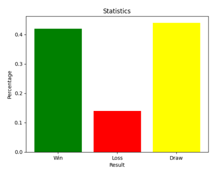
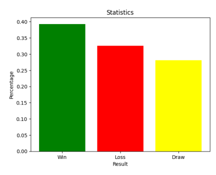
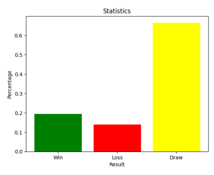
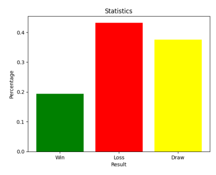
	No Symmetries	Symmetries
Training		
Random Performances		
My Player Performances		

Table 2: Comparisons in exploiting symmetries

Results :

The agent seems to have learned how to block most of the opponent's moves, but it still has some problems in winning. Symmetries can be exploited to propagate the knowledge acquired to other different (but symmetric) states, but unfortunately, performances decrease.

3 Peer Reviews :

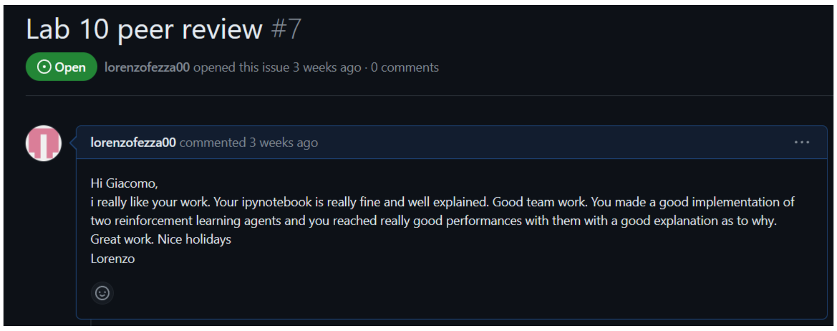
During the course, laboratories were accompanied by peer reviews . This activity, carried out for the last three laboratories and for two colleagues for each lab, proved to be not only interesting and stimulating but also immensely helpful in fostering a collaborative learning environment.

This process provided a valuable opportunity to assess and analyze the coding approaches taken by others, enabling me to improve my own level. The diverse perspectives presented in the peer-reviewed codes expanded my understanding of different problem-solving strategies and allowed me to consider alternative approaches.

The feedback received through the peer reviews was really useful in refining my coding skills. Constructive criticism highlighted potential improvements. Additionally, positive feedback served as encouragement and reinforcement of effective coding practices.

To illustrate this collaborative learning experience, I have included screenshots of the peer reviews I conducted and received. These snapshots capture the constructive dialogue that took place, showcasing the communal effort to enhance our collective understanding of computational intelligence concepts.

In conclusion, the incorporation of peer reviews not only enriched my learning experience but also provided a platform for constructive feedback, fostering a culture of collaboration within the Computational Intelligence course.

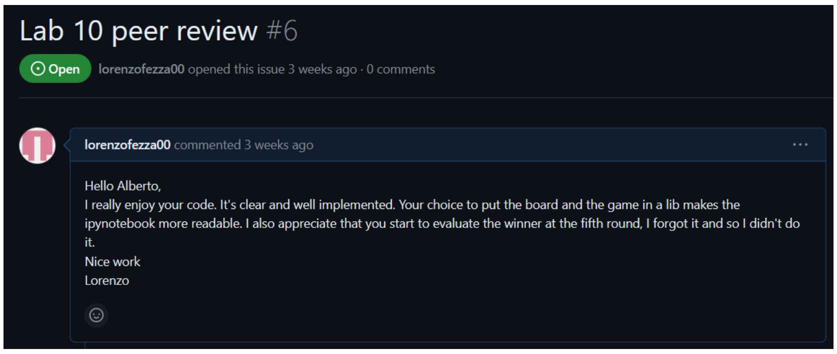


Lab 10 peer review #7

Open lorenzofezza00 opened this issue 3 weeks ago · 0 comments

lorenzofezza00 commented 3 weeks ago

Hi Giacomo,
i really like your work. Your ipynotebook is really fine and well explained. Good team work. You made a good implementation of two reinforcement learning agents and you reached really good performances with them with a good explanation as to why.
Great work. Nice holidays
Lorenzo

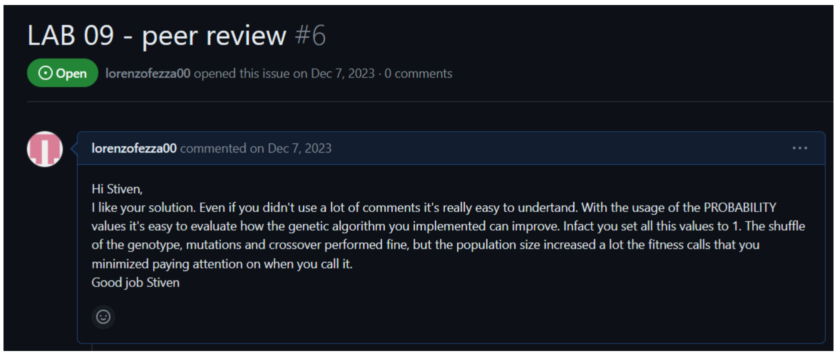


Lab 10 peer review #6

Open lorenzofezza00 opened this issue 3 weeks ago · 0 comments

lorenzofezza00 commented 3 weeks ago

Hello Alberto,
I really enjoy your code. It's clear and well implemented. Your choice to put the board and the game in a lib makes the ipynotebook more readable. I also appreciate that you start to evaluate the winner at the fifth round, I forgot it and so I didn't do it.
Nice work
Lorenzo

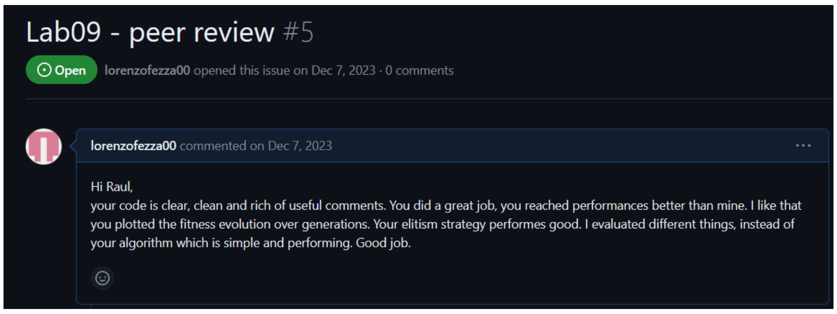


LAB 09 - peer review #6

Open lorenzofezza00 opened this issue on Dec 7, 2023 · 0 comments

lorenzofezza00 commented on Dec 7, 2023

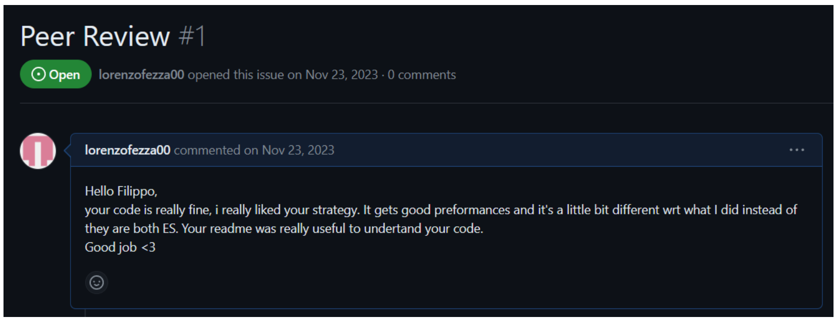
Hi Stiven,
I like your solution. Even if you didn't use a lot of comments it's really easy to undertand. With the usage of the PROBABILITY values it's easy to evaluate how the genetic algorithm you implemented can improve. Infact you set all this values to 1. The shuffle of the genotype, mutations and crossover performed fine, but the population size increased a lot the fitness calls that you minimized paying attention on when you call it.
Good job Stiven


Lab09 - peer review #5

Open lorenzofezza00 opened this issue on Dec 7, 2023 · 0 comments

lorenzofezza00 commented on Dec 7, 2023

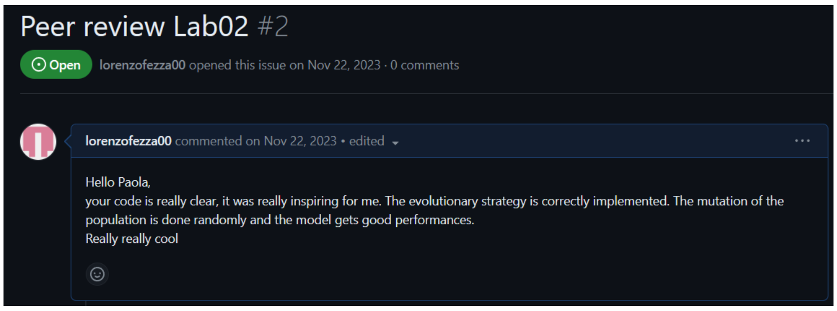
Hi Raul,
your code is clear, clean and rich of useful comments. You did a great job, you reached performances better than mine. I like that you plotted the fitness evolution over generations. Your elitism strategy performes good. I evaluated different things, instead of your algorithm which is simple and performing. Good job.


Peer Review #1

Open lorenzofezza00 opened this issue on Nov 23, 2023 · 0 comments

lorenzofezza00 commented on Nov 23, 2023

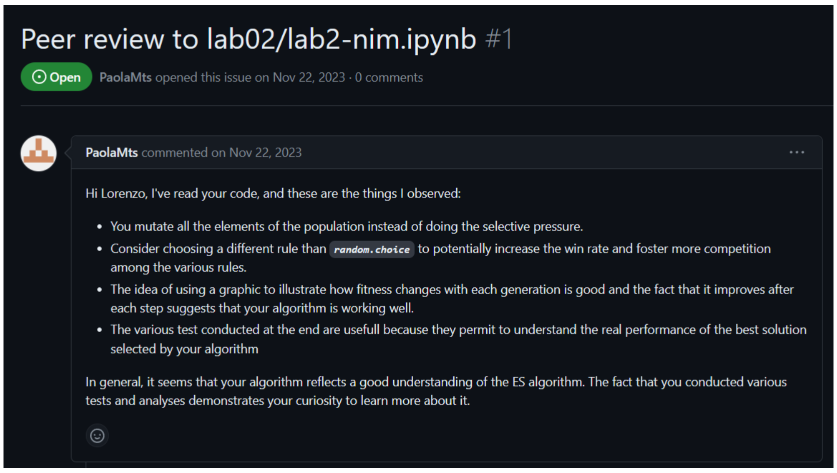
Hello Filippo,
your code is really fine, i really liked your strategy. It gets good preformances and it's a little bit different wrt what I did instead of they are both ES. Your readme was really useful to undertand your code.
Good job <3


Peer review Lab02 #2

Open lorenzofezza00 opened this issue on Nov 22, 2023 · 0 comments

lorenzofezza00 commented on Nov 22, 2023 • edited

Hello Paola,
your code is really clear, it was really inspiring for me. The evolutionary strategy is correctly implemented. The mutation of the population is done randomly and the model gets good performances.
Really really cool


Peer review to lab02/lab2-nim.ipynb #1

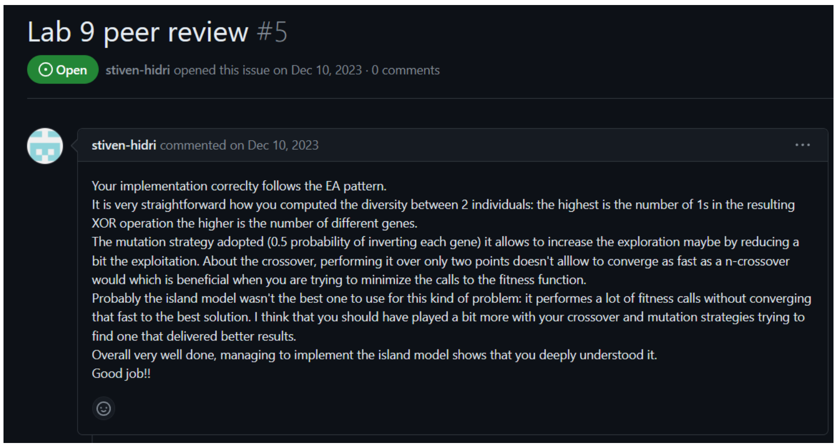
Open PaolaMts opened this issue on Nov 22, 2023 · 0 comments

PaolaMts commented on Nov 22, 2023

Hi Lorenzo, I've read your code, and these are the things I observed:

- You mutate all the elements of the population instead of doing the selective pressure.
- Consider choosing a different rule than `random.choice` to potentially increase the win rate and foster more competition among the various rules.
- The idea of using a graphic to illustrate how fitness changes with each generation is good and the fact that it improves after each step suggests that your algorithm is working well.
- The various test conducted at the end are usefull because they permit to understand the real performance of the best solution selected by your algorithm

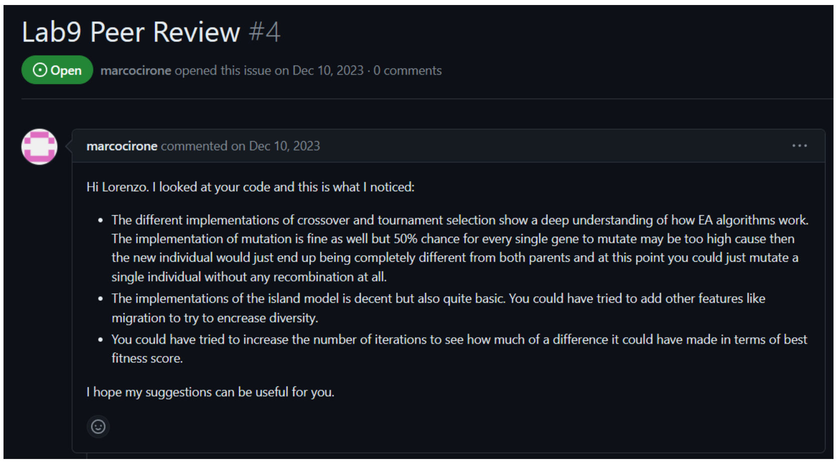
In general, it seems that your algorithm reflects a good understanding of the ES algorithm. The fact that you conducted various tests and analyses demonstrates your curiosity to learn more about it.


Lab 9 peer review #5

Open stiven-hidri opened this issue on Dec 10, 2023 · 0 comments

stiven-hidri commented on Dec 10, 2023

Your implementation correclty follows the EA pattern.
It is very straightforward how you computed the diversity between 2 individuals: the highest is the number of 1s in the resulting XOR operation the higher is the number of different genes.
The mutation strategy adopted (0.5 probability of inverting each gene) it allows to increase the exploration maybe by reducing a bit the exploitation. About the crossover, performing it over only two points doesn't alllow to converge as fast as a n-crossover would which is beneficial when you are trying to minimize the calls to the fitness function.
Probably the island model wasn't the best one to use for this kind of problem: it performes a lot of fitness calls without converging that fast to the best solution. I think that you should have played a bit more with your crossover and mutation strategies trying to find one that delivered better results.
Overall very well done, managing to implement the island model shows that you deeply understood it.
Good job!!


Lab9 Peer Review #4

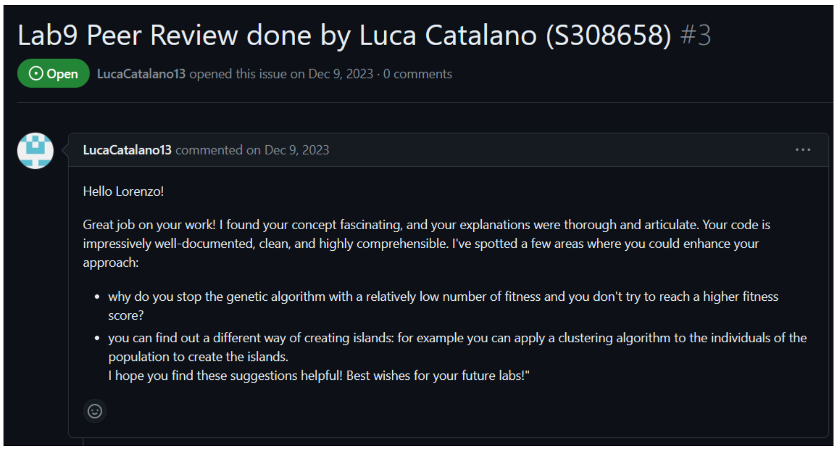
Open marcocirone opened this issue on Dec 10, 2023 · 0 comments

marcocirone commented on Dec 10, 2023

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.


Lab9 Peer Review done by Luca Catalano (S308658) #3

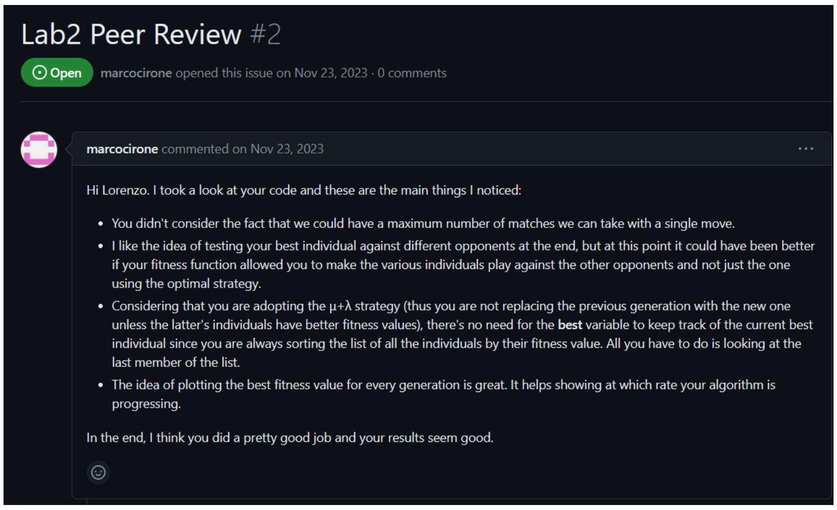
Open LucaCatalano13 opened this issue on Dec 9, 2023 · 0 comments

LucaCatalano13 commented on Dec 9, 2023

Hello Lorenzo!

Great job on your work! I found your concept fascinating, and your explanations were thorough and articulate. Your code is impressively well-documented, clean, and highly comprehensible. I've spotted a few areas where you could enhance your approach:

- why do you stop the genetic algorithm with a relatively low number of fitness and you don't try to reach a higher fitness score?
- you can find out a different way of creating islands: for example you can apply a clustering algorithm to the individuals of the population to create the islands.
  I hope you find these suggestions helpful! Best wishes for your future labs!"


Lab2 Peer Review #2

Open marcocirone opened this issue on Nov 23, 2023 · 0 comments

marcocirone commented on Nov 23, 2023

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.