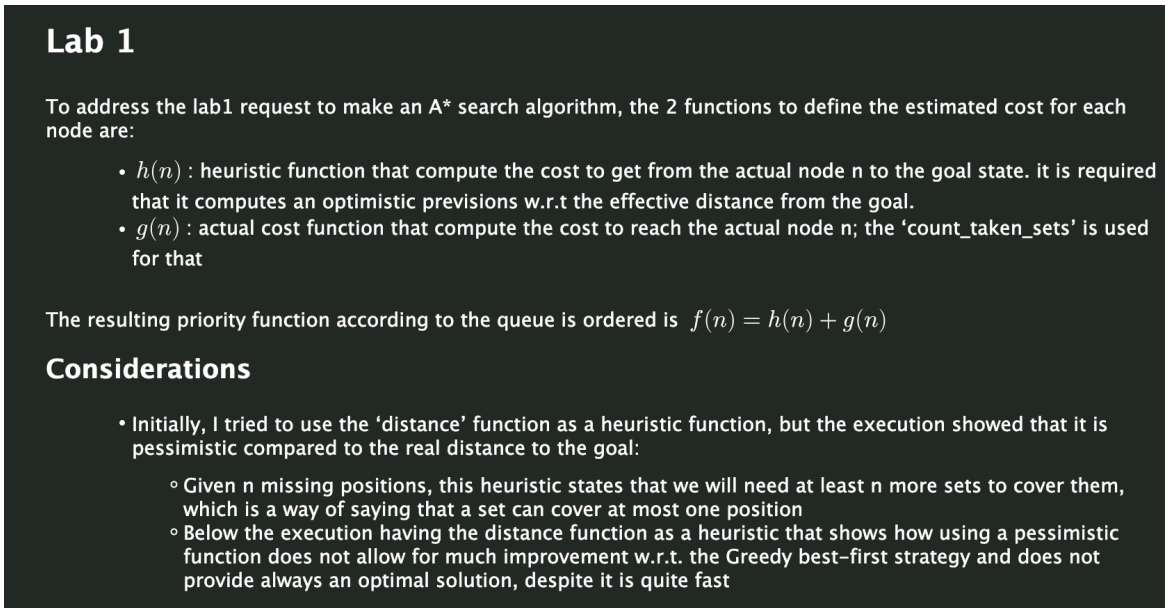
Report

Computational Intelligence Course

Giorgio Cacopardi

Lab 1

- Firstly I take a look to the notes taken during the lecture and to some online resources to make an idea and better comprehend the problem
- Then I started to develop the lab, using as base the code of the professor, reorganizing it and trying to make a single function adaptable to many strategies instead of developing different functions for different strategies



- Initially I developed, based also on the code of my colleagues, a wrong heuristic function due to its pessimistic considerations consisting in calculate the distance from the actual set to the goal

```
def distance(state):
    return PROBLEM_SIZE - sum(covered(state))
```

- After some days, I realized, thanks to the professor, that the functions is not suited for the tasks so, using as base the functions provided by the professor, I slightly improved the h3 function to reduce some computation and increase its efficiency, specifying that the improvement is appreciable only in case of large sets and problem size

```python
def h3(state):
    already_covered = covered(state)
    if np.all(already_covered):
        return 0
    missing_size = PROBLEM_SIZE - sum(already_covered)
    candidates = sorted((sum(np.logical_and(s, np.logical_not(already_covered))) for s in SETS),
     reverse=True)
    taken = 1
    while sum(candidates[:taken]) < missing_size:
        taken += 1
    return taken

def h3_improved(state):
    already_covered = covered(state)
    if np.all(already_covered):
        return 0
    missing_size = PROBLEM_SIZE - sum(already_covered)
    candidates = sorted((sum(np.logical_and(SETS[s], np.logical_not(already_covered))) for s in
     state.not_taken), reverse=True)
    taken = 1
    while sum(candidates[:taken]) < missing_size:
        taken += 1
    return taken
```

```
A* search with h3
Solved in 664 steps (4 tiles)
Solution: State(taken={81, 66, 74, 5}, not_taken={0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 78, 79, 80, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99})
Execution time of A* search with h3 not improved: 19.107330083847046
A* search with h3 improved
Solved in 664 steps (4 tiles)
Solution: State(taken={81, 66, 74, 5}, not_taken={0, 1, 2, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 67, 68, 69, 70, 71, 72, 73, 75, 76, 77, 78, 79, 80, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99})
Execution time of A* search with h3 improved: 18.70634698867798
```

- The improvement lead to a minor computation time w.r.t. to the old h3 despite the result is the same for both. Obviously to be the improvement remarkable is necessary to test over very larges PROBLEM_SIZES and NUM_SETS

Halloween Challenge

- For the Halloween challenge I tried to develop 3 different Hill Climbing approach to compute the solutions: with Termination, with Random Restart and with Steepest Step
- In the begin, I develop an algorithm that does not stop and try to increase always the solution
- Then I realized that this way was useless since there are local maxima where the algorithm get stuck so I put a termination condition to stop in the case of local maxima

```
def hill_climbing_with_termination(fitness_func, state_current, no_improvement_iterations, problem_size, sets):
    counter = 0
    fitness_call = 0
    no_improvements = 0
    iterations = 0
    current_best_state_fitness = fitness_func(state_current, problem_size, sets)
    for step in range(10000):
        iterations += 1
        if no_improvements >= no_improvement_iterations:
            break
        new_state = tweak(state_current, problem_size)
        new_state_fitness = fitness_func(new_state, problem_size, sets)
        fitness_call += 1
        if  new_state_fitness > current_best_state_fitness:
            counter += 1
            no_improvements = 0
            state_current = new_state
            current_best_state_fitness = new_state_fitness
        else:
            no_improvements += 1
    return state_current, current_best_state_fitness, fitness_call
```

- To improve this solution I tried to make some tests with different value to compute the best combination

```
list_best_solutions = []
for size in PROBLEM_SIZE:
    for density in DENSITY:
        best_solution = None
        for max_iter in range(size//10, size+1, size//10):
            SETS = make_set_covering_problem(size, size, density)
            current_state = [choice([False, False, False, False, False, False]) for _ in range(size)]
            state, fitness_state, fitness_call = hill_climbing_with_termination(fitness, current_state,
             max_iter, size, SETS)
            if best_solution is None or (fitness_state > best_solution[0]) or (fitness_state ==
             best_solution[0] and fitness_call < best_solution[1]):
                best_solution = (fitness_state, fitness_call, max_iter, size, density)
        list_best_solutions.append(best_solution)
```

- As second algorithm I tried to improve the previous and instead of stopping the algorithm after some iteration without improvements, I make a random restart to escape from local maxima

```python
def hill_climbing_with_random_restart(fitness_func, state_current, no_improvement_iterations, problem_size, sets, prob):
    counter = 0
    fitness_call = 0
    no_improvements = 0
    iterations = 0
    current_best_state_fitness = fitness_func(state_current, problem_size, sets)
    current_best_state = copy(state_current)
    for step in range(10000):
        iterations += 1
        if no_improvements >= no_improvement_iterations:
            current_best_state = copy(state_current)
            state_current = [random() < prob for _ in range(size)]
        new_state = tweak(state_current, problem_size)
        new_state_fitness = fitness_func(new_state, problem_size, sets)
        fitness_call += 1
        if  new_state_fitness > current_best_state_fitness:
            counter += 1
            no_improvements = 0
            state_current = new_state
            current_best_state = copy(state_current)
            current_best_state_fitness = new_state_fitness
        else:
            no_improvements += 1
    return current_best_state, current_best_state_fitness, fitness_call
```

- I always make some iterations to find both the max iterations parameter and the probability to make the random restart to find the better solution
- I noticed that with the except of the case with PROBLEM_SIZE = 100, the algorithm not only does not improve the solution but also makes a lot more iterations
- As final algorithm I tried to implement a steepest step approach, following the same procedure
- The algorithm actually improved the solutions but sacrifice a lot in terms of number of fitness function call

```python
def steepest_step(fitness, state, size, sets):
    new_state = tweak_index(state, 0)
    new_state_fitness = fitness(new_state, size, sets)
    for index in range(0, size):
            temp_state = tweak_index(state, index)
            temp_state_fitness = fitness(temp_state, size, sets)
            if temp_state_fitness > new_state_fitness:
                new_state = temp_state
                new_state_fitness = temp_state_fitness
    return new_state, new_state_fitness

def hill_climbing_steepest_step(fitness_func, state_current, no_improvement_iterations, problem_size, sets, prob):
    counter = 0
    fitness_call = 0
    no_improvements = 0
    current_best_state_fitness = fitness_func(state_current, problem_size, sets)
    for step in range(1000):
        if no_improvements >= no_improvement_iterations:
            break
        a = random() < prob
        if not a:
            new_state, new_state_fitness = steepest_step(fitness, state_current, problem_size, sets)
            fitness_call += problem_size
        else:
            new_state = tweak(state_current, problem_size)
            new_state_fitness = fitness_func(new_state, problem_size, sets)
            fitness_call += 1
        if new_state_fitness > current_best_state_fitness:
            counter += 1
            no_improvements = 0
            state_current = new_state
            current_best_state = copy(state_current)
            current_best_state_fitness = new_state_fitness
        else:
            no_improvements += 1
    return current_best_state, current_best_state_fitness, fitness_call
```

- The final table collecting all the results:

	PROBLEM_SIZE	DENSITY	FITNESS_EVALUATION	FITNESS_CALL
Hill climbing with early termination	100	0.3	(100, -10)	31
	100	0.7	(100, -4)	14
	1000	0.3	(1000, -14)	2681
	1000	0.7	(1000, -6)	509
	5000	0.3	(5000, -18)	2114
	5000	0.7	(5000, -8)	2659
	PROBLEM_SIZE	DENSITY	FITNESS_EVALUATION	FITNESS_CALL
Hill climbing with random restart	100	0.3	(100, -7)	10000
	100	0.7	(100, -3)	10000
	1000	0.3	(1000, -14)	10000
	1000	0.7	(1000, -6)	10000
	5000	0.3	(5000, -18)	10000
	5000	0.7	(5000, -8)	10000
	PROBLEM_SIZE	DENSITY	FITNESS_EVALUATION	FITNESS_CALL
Hill climbing with steepest step	100	0.3	(100, -6)	1206
	100	0.7	(100, -3)	1004
	1000	0.3	(1000, -12)	62050
	1000	0.7	(1000, -4)	50054

Lab 2

- Initially I had some difficulties to understand the delivery, I tried to read some online sources to better understand how to model the problem
- Then, based on some colleagues suggestions I tried to implement a possible solution that consist in make an evolutionary algorithm to establish which of the strategies can be better

```python
# Evolutionary parameters
POPULATION_SIZE = 30
MUTATION_RATE = 0.2
NUMBER_GENERATIONS = 50
NIM_ROWS = 10
STRATEGIES = [gabriele, pure_random, optimal, adaptive]
AGENT_STRATEGIES = 8

def generate_random_agent():

    agent_strategies =  [random.choices(STRATEGIES, weights=[random.randint(1, 4) for _ in range(len(STRATEGIES))])[0]
     for _ in range(AGENT_STRATEGIES)]
    strategies_weights = random.choices([1, 2, 3, 4], k=AGENT_STRATEGIES)
    return (agent_strategies, strategies_weights)


def fitness(agent, number_of_matches=20):
    victories_agent = 0
    for i in range(number_of_matches):
        number_of_rows = random.randint(2, 10)
        max_k = random.randint(2, 10)
        nim = Nim(NIM_ROWS, max_k)
        results = play_game_against_expert(nim, agent)
        victories_agent += 1 if results == 1 else 0

    return victories_agent/number_of_matches


def mutate(agent):
    strategies, weights = agent
    mutated_strategies = strategies[:]
    idx_to_mutate = random.randint(0, len(mutated_strategies) - 1)
    mutated_strategies[idx_to_mutate] = random.choice(STRATEGIES)
    mutated_weights = weights[:]
    mutated_weights[random.randint(0, len(mutated_strategies) -1 )] = random.choice([1, 2, 3, 4])
    return mutated_strategies, mutated_weights


def reproduce(agent1, agent2):
    strategies1, weights1 = agent1
    strategies2, weights2 = agent2
    crossover_point = random.randint(0, len(strategies1))
    child_strategies = strategies1[:crossover_point] + strategies2[crossover_point:]
    child_weights = weights1[:crossover_point] + weights2[crossover_point:]
    return child_strategies, child_weights
```

```python
# Initialize the population
population = [generate_random_agent() for _ in range(POPULATION_SIZE - 1)]

# Evolutionary loop
for generation in range(NUMBER_GENERATIONS):
    # Evaluate current generation
    fitness_scores = [fitness(agent) for agent in population]

    if generation % 5 == 0:
        max_fitness = max(fitness_scores)
        print(f"- Generation: {generation} - Best Fitness: {max_fitness} - Avg Fitness: {sum(fitness_scores) / len(fitness_scores)}")

    # Keep the best agent from the previous generation
    best_of_generation = max(population, key=fitness)

    # Select parents
    best_indexes = sorted(range(len(fitness_scores)), key=lambda x: fitness_scores[x], reverse=True)[:POPULATION_SIZE // 3]
    selected_parents = [population[index] for index in best_indexes]
    # Create next generation
    new_population = [best_of_generation]

    for i in range(POPULATION_SIZE - 1):
        if random.random() < MUTATION_RATE:
            new_population.append(mutate(random.choice(selected_parents)))
        else:
            agent1 = random.choice(selected_parents)
            agent2 = random.choice(selected_parents)
            new_population.append(reproduce(agent1, agent2))

    population = new_population

# Print the best agents
best_3_agents = sorted(population, key=fitness, reverse=True)[:3]
for i, agent in enumerate(best_3_agents):
    print(f"Agent {i} using functions: {[func.__name__ for func in agent[0]]}, weights: {agent[1]} with a fitness value of: {fitness(agent)}")
```

- As additional strategy to include in the cycle I create an adaptive strategy that initially were incorrect because of a bug I did not noticed in the final move where it execute moves by no taking elements so the strategy always wins against the expert agent, but I don't noticed the problem

```python
def adaptive(state: Nim) -> Nimply:
    """A strategy that can adapt its parameters"""
    remaining_moves = sum(state.rows) - 1;
    tot_moves = sum([i * 2 + 1 for i in range(NIM_ROWS)])
    genome = remaining_moves/tot_moves

    index_max_rows = state.rows.index(max(state.rows))
    objects_to_take = min(int(genome * max(state.rows)), state.k)
    return Nimply(index_max_rows, objects_to_take)
```

- 2 days later I review the code and noticed the error so I fix it and I realize that the simple adaptive strategy were not actually so good but still better than the other one

```python
def adaptive(state: Nim) -> Nimply:
    """A strategy that can adapt its parameters"""
    remaining_moves = sum(state.rows) - 1;
    tot_moves = sum([i * 2 + 1 for i in range(NIM_ROWS)])
    genome = remaining_moves/tot_moves

    index_max_rows = state.rows.index(max(state.rows))
    objects_to_take = max(min(int(genome * max(state.rows)), state.k), 1)
    return Nimply(index_max_rows, objects_to_take)
```

- The final results weren't so good because all the strategies lose for the most part of the time against the optimal strategy

```
 - Generation: 0 - Best Fitness: 0.15 - Avg Fitness: 0.03620689655172414
 - Generation: 5 - Best Fitness: 0.2 - Avg Fitness: 0.060000000000000026
 - Generation: 10 - Best Fitness: 0.15 - Avg Fitness: 0.05333333333333335
 - Generation: 15 - Best Fitness: 0.2 - Avg Fitness: 0.07000000000000002
 - Generation: 20 - Best Fitness: 0.25 - Avg Fitness: 0.08333333333333333
 - Generation: 25 - Best Fitness: 0.2 - Avg Fitness: 0.06333333333333337
 - Generation: 30 - Best Fitness: 0.25 - Avg Fitness: 0.06500000000000002
 - Generation: 35 - Best Fitness: 0.2 - Avg Fitness: 0.055000000000000014
 - Generation: 40 - Best Fitness: 0.2 - Avg Fitness: 0.06666666666666668
 - Generation: 45 - Best Fitness: 0.25 - Avg Fitness: 0.09499999999999999
Agent 0 using functions: ['pure_random', 'pure_random', 'optimal', 'pure_random', 'pure_random', 'gabriele', 'pure_random', 'pure_random'],
 weights: [1, 3, 4, 2, 1, 4, 2, 3] with a fitness value of: 0.05
Agent 1 using functions: ['pure_random', 'pure_random', 'optimal', 'pure_random', 'pure_random', 'adaptive', 'pure_random', 'pure_random'],
 weights: [3, 3, 4, 2, 1, 3, 2, 1] with a fitness value of: 0.05
Agent 2 using functions: ['pure_random', 'pure_random', 'optimal', 'pure_random', 'pure_random', 'pure_random', 'pure_random',
'pure_random'], weights: [1, 3, 4, 2, 1, 4, 2, 3] with a fitness value of: 0.15

From the results the most used strategy is the random one, that show that the agents developed do not lead to good results
To test it, here there are 4 simulations to test all the four strategies against the expert agent to confirm the result
```

- I make the reviews for:
  - Laura Amoroso s313813: https://github.com/AmorosoLaura/computational_intelligence.git

GioC1810 commented on Nov 21, 2023

Hi Laura,
First of all i appreciate your code organization and readability, it was very smooth to read and understand.
I would like to give you some suggestion about how to improve your code, especially about the evolutionary part:

1. About your "evolving_strategy" function i suggest to improve readability and conciseness by structure your code like the image below so that in this way there are less code lines, you can avoid to check if a strategy is in the dictionary or have to be added thanks to the defaultdict dictionary of the collections package:

```
def evolving_strategy_2(state: Nim, w) -> Nimply:
    move_weights = defaultdict(list)

    # Collect the suggested moves for each strategy along with their weights
    moves = [s(state) for s in strategies]
    for i, move in enumerate(moves):
        move_weights[move].append(w[i])

    # Calculate the voting based on moves and their associated weights
    voting = {move: sum(weights) + len(weights) for move, weights in move_weights.items()}
    print(voting)
    # Find the move with the maximum total weight
    max_key = max(voting, key=voting.get)
    return max_key
```

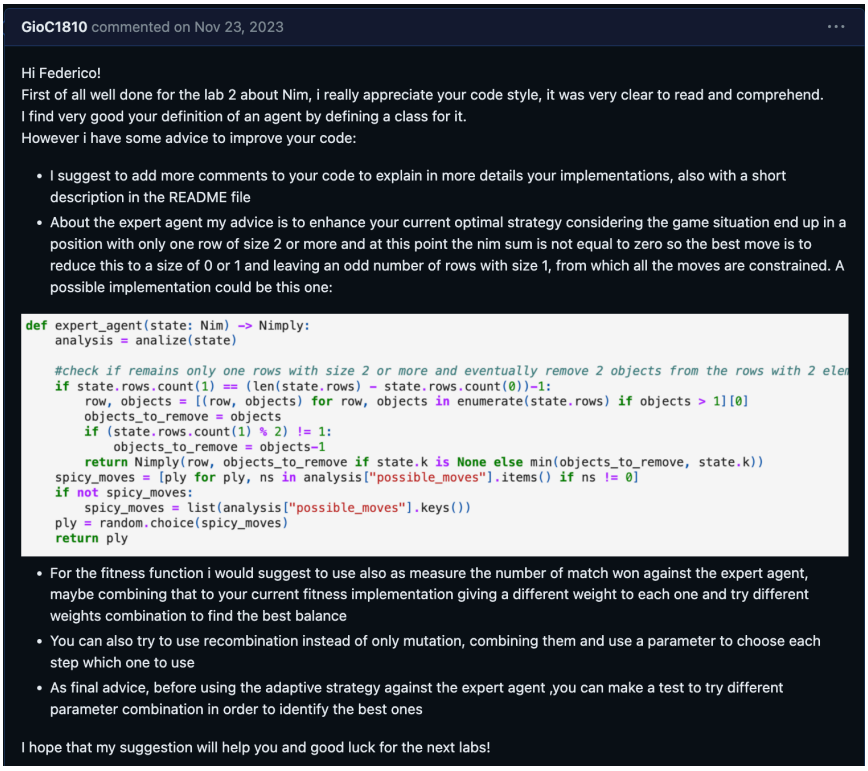
2. About your "fitness" function my advice is to refactor it in this way to optimize it and reducing redundant function calls:

```
def fitness_2(w):
    """Computes how many games are won against 3 players"""
    counter = 0

    # Pre-calculate the matches with each player's strategy
    matches = {
        'expert_agent': [match(evolving_strategy, expert_agent, w) for _ in range(num_eras)],
        'pure_random': [match(evolving_strategy, pure_random, w) for _ in range(num_eras)],
        'gabriele': [match(evolving_strategy, gabriele, w) for _ in range(num_eras)],
    }

    # Calculate the wins based on predefined matches
    for era in range(num_eras):
        if era < num_eras / 2:
            counter += sum(matches[player][era] == 0 for player in matches)
        else:
            counter += sum(matches[player][era] == 1 for player in matches)

    return counter
```

3. I found really appropriate to adopt a gaussian distribution to change the weights in your evolutionary strategy, in addition to that i suggest to explore also the recombination approach and not only the mutation to see if this can improve your algorithm

I hope that my suggestion can help you

  - Federico Buccellato s309075: https://github.com/FedeBucce/Computational_intelligence.git



GioC1810 commented on Nov 23, 2023

Hi Federico!
First of all well done for the lab 2 about Nim, i really appreciate your code style, it was very clear to read and comprehend.
I find very good your definition of an agent by defining a class for it.
However i have some advice to improve your code:

- I suggest to add more comments to your code to explain in more details your implementations, also with a short description in the README file
- About the expert agent my advice is to enhance your current optimal strategy considering the game situation end up in a position with only one row of size 2 or more and at this point the nim sum is not equal to zero so the best move is to reduce this to a size of 0 or 1 and leaving an odd number of rows with size 1, from which all the moves are constrained. A possible implementation could be this one:

```
def expert_agent(state: Nim) -> Nimply:
    analysis = analize(state)

    #check if remains only one rows with size 2 or more and eventually remove 2 objects from the rows with 2 ele
    if state.rows.count(1) == (len(state.rows) - state.rows.count(0))-1:
        row, objects = [(row, objects) for row, objects in enumerate(state.rows) if objects > 1][0]
        objects_to_remove = objects
        if (state.rows.count(1) % 2) != 1:
            objects_to_remove = objects-1
        return Nimply(row, objects_to_remove if state.k is None else min(objects_to_remove, state.k))
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].items() if ns != 0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply
```

- For the fitness function i would suggest to use also as measure the number of match won against the expert agent, maybe combining that to your current fitness implementation giving a different weight to each one and try different weights combination to find the best balance
- You can also try to use recombination instead of only mutation, combining them and use a parameter to choose each step which one to use
- As final advice, before using the adaptive strategy against the expert agent ,you can make a test to try different parameter combination in order to identify the best ones

I hope that my suggestion will help you and good luck for the next labs!

Lab 3

- During the first implementation I try to tune with a lot of parameters to find the best combination but this required a lot of time in computation
- So, i decide to reduce the number of parameters and reduce the initial populations size and number of generations to first find the possible best parameters and then use bigger population size and generations number to actually try to test the algorithm

Parameters tuning

This code has the purpose to tune some parameters, considering a small populations size and generations number, to find the best combination that will be used then to test the algorithm
The parameters are:

- OFFSPRING_SIZE: It indicates the size of the generated offspring
- CROSS_OPERATIONS: It contains an array of 3 integer, each one corresponding to a different cross function
- NUMBER_CROSSOVER_POINTS: It contains an array of integer that indicates the point in which the parents will be split in the crossover_n_point function
- TOURNAMENT_SIZE: It indicates how much is the size of the array of individuals selected to compete in the tournament selection
- MUTATION_POINTS: It indicates how much points the mutation change

```python
import itertools

LOCI = 1000
PROBLEM_INSTANCES = [1, 2, 5, 10]

GENERATIONS_TEST = 1000
POPULATIONS_NUMBER_TEST = 50

OFFSPRING_SIZE = 20
NUMBER_CROSSOVER_POINTS = 2
MUTATION_RATES = {1: 0.8, 2: 0.2, 5: 0.2, 10: 0.2}

TOURNAMENT_SIZE = [2, 4, 8]
SELECTION_TYPE = [0, 1]
CROSS_OPERATIONS = [0, 1, 2]
MUTATION_POINTS = [1, 4]

parameter_combinations = list(itertools.product(PROBLEM_INSTANCES, TOURNAMENT_SIZE, SELECTION_TYPE, MUTATION_POINTS, CROSS_OPERATIONS))

best_parameters = {}
best_individuals = {}

for instance, t_size, s_type, m_points, cr_op in parameter_combinations:
    operator_agent = Operators_agent(NUMBER_CROSSOVER_POINTS, m_points, MUTATION_RATES[instance])
    individual, calls = launch_es_cycle(instance, POPULATIONS_NUMBER_TEST, LOCI, GENERATIONS_TEST, OFFSPRING_SIZE, t_size, s_type, operator_agent,
     cr_op)
    if instance not in best_parameters or (individual.fitness > best_individuals[instance][0].fitness)\
            or (math.isclose(individual.fitness, best_individuals[instance][0].fitness) and calls < best_individuals[instance][1]):
        best_individuals[instance] = [individual, calls]
        best_parameters[instance] = [t_size, s_type, m_points, cr_op]

for instance in PROBLEM_INSTANCES:
    print(f"Instance {instance}\n "
          f"Best Individual: Score: {best_individuals[instance][0].fitness}\n "
          f"Calls: {best_individuals[instance][1]}")
    print(f"Parameters:\n "
          f"Tournament size:{best_parameters[instance][0]}\n"
          f"Selection type:{best_parameters[instance][1]}\n"
          f"Points to mutate:{best_parameters[instance][2]}\n"
          f"Cross operation:{best_parameters[instance][3]}\n")
```

- The first attempts wasn't very good so I try to reduce the parameters to tune and choose based on previous computations the best ones
- For the others parameters I try to vary more the range of values they can assume to see if one of them can lead to better improvement

- I repeat the tuning but I also modify the function for the evolutionary cycle by putting a stop condition when an individual reaches the max fitness scores, to avoid useless fitness calls where they are not necessary and also by putting a tournament selection to select the parent for reproduction instead of the previous method that was more deterministic
- It improve slightly but not as expected so I decided to change the mutation function to dynamically adapt the points to change based on the actual fitness score of the individual, in a way that higher fitness score lead to minor points to mutate
- I also set a dynamical mutation rate change that depends on the average of the fitnesses scores of the generation, higher fitness scores lead to higher mutation rate
- After I realized I make a mistake about the muting rate, since I switch in the code the probabilities and mutation rate actually refers to recombination rate, so after fixing it I finally solved the problem with instance 1
- Then, for the others I build an operator agent that execute mutation and recombination and keep track on the total number of crossover and mutation to compute a probabilities weight vector to reward more the recombination operation that produce child with better fitness, but it was difficult to compute how much increment the probability of the vector so I abandoned this idea

- In the end, by take inspiration by some colleagues code, I add to the original evolutionary cycle function a control on the standard deviation in a way that when it drops above a certain thresholds, I delete some parents using a stochastic universal sampling, and I re insert random individuals to the population to reach again the original population number; this procedures help to reach good results for the problem instance 2, and also slightly improve the instance 5
- For the instance 10, I tried some combinations of solutions but nothing really work so I just tried to increment the generations but with not good result
- About the code I refactoring it to have an operator agent which includes the principal operator like mutation and recombination and that allows to compute some statistics about how much the mutation or the recombination help in improving the individual fitness
- For the parent selection before generating the offspring I used the stochastic universal sampling that produces a better sample of the distribution compared to the roulette wheel

```python
def stochastic_universal_sampling(population: List[Individual], parent_selection_rate: int) -> List[Individual]:

    total_fitness = sum(ind.fitness for ind in population)
    pointer_distance = total_fitness / len(population)
    num_selected_parents = int(parent_selection_rate)
    start = uniform(0, pointer_distance)
    pointers = [start + i * pointer_distance for i in range(num_selected_parents)]

    new_population = []
    current_index = 0
    for pointer in pointers:
        while pointer > 0:
            pointer -= population[current_index].fitness
            current_index = (current_index + 1) % len(population)
        new_population.append(population[current_index])
        if len(new_population) == num_selected_parents:
            break

    return new_population
```

- For selecting the parent that contributes to the offspring generation I used a classical tournament selection

- For the offspring generation I tried 2 different functions, one that computes or mutation or recombination, the other that mix both

```python
def offspring_generation(parents: List[Individual], offspring_size: int, operator_agent: "Operators_agent", tournament_size: int,
 cross_operation) -> List[Individual]:
    offspring = []
    offspring_length = int(offspring_size)

    for _ in range(offspring_length):
        p = select_parent(parents, tournament_size)
        if random.random() < operator_agent.mutation_rate:
            offspring.append(operator_agent.mutate(p))
        else:
            p2 = select_parent(parents, tournament_size)
            offspring.append(operator_agent.crossover(p, p2, cross_operation))

    return offspring
```

Executed at 2023.12.14 15:24:29 in 2ms

```python
def offspring_generation_mixture(parents: List[Individual], offspring_size: int, operator_agent: "Operators_agent", tournament_size: int,
 cross_operation) -> List[Individual]:
    offspring = []
    offspring_length = int(offspring_size)

    for _ in range(offspring_length):
        p = select_parent(parents, tournament_size)
        p2 = select_parent(parents, tournament_size)
        if random.random() < operator_agent.mutation_rate:
            offspring.append(operator_agent.mutate(operator_agent.crossover(p, p2, cross_operation)))
        else:

            offspring.append(operator_agent.crossover(p, p2, cross_operation))

    return offspring
```

- I used different functions to compute the distance between individuals. The distance has the purpose to indicates if the individuals became too similar among them

```python
def hamming_distance(ind1: "Individual", ind2: "Individual") -> float:
    distance = 0
    for i in range(ind1.n_loci):
        if ind1.genome[i]!= ind2.genome[i]:
            distance += 1
    return distance/len(ind1.genome)

def jaccard_distance(ind1: "Individual", ind2: "Individual") -> float:
    intersection = np.logical_and(ind1.genome, ind2.genome).sum();
    union = np.logical_or(ind1.genome, ind2.genome).sum();
    return 1-(intersection/union)

def euclidean_distance(ind1: "Individual", ind2: "Individual") -> float:
    distance = sum((int(ind1.genome[i]) - int(ind2.genome[i]))**2 for i in range(ind1.n_loci))
    max_possible_distance = (2 ** 0.5) * ind1.n_loci  # Maximum possible distance for normalized Euclidean distance
    normalized_distance = (distance ** 0.5) / max_possible_distance
    return normalized_distance

def manhattan_distance_binary(ind1: "Individual", ind2: "Individual") -> float:

    distance = sum(abs(int(ind1.genome[i]) - int(ind2.genome[i])) for i in range(ind1.n_loci))
    max_possible_distance = ind1.n_loci  # Maximum possible distance for normalized Manhattan distance
    normalized_distance = distance / max_possible_distance
    return normalized_distance
```

- In the end I make 3 different algorithm for the evolutionary cycle
  - Standard cycle

```
def launch_es_cycle(problem_instance: int, populations_number: int, n_loci: int, generations: int, offspring_size: int, tournament_size: int,
 selection_type: int, operator_agent: "Operators_agent", cross_operation: int):

    fitness_func = lab9_lib.make_problem(problem_instance)
    population = [Individual(fitness_func, n_loci) for _ in range(populations_number)]

    for gen in range(generations):

        offspring = offspring_generation(population, offspring_size, operator_agent, tournament_size, cross_operation)
        population = sorted(population+offspring if selection_type == 1 else offspring, key=lambda p: p.fitness, reverse=True)[:populations_number]
        if math.isclose(1, population[0].fitness):
            break

    return population[0], fitness_func.calls
```

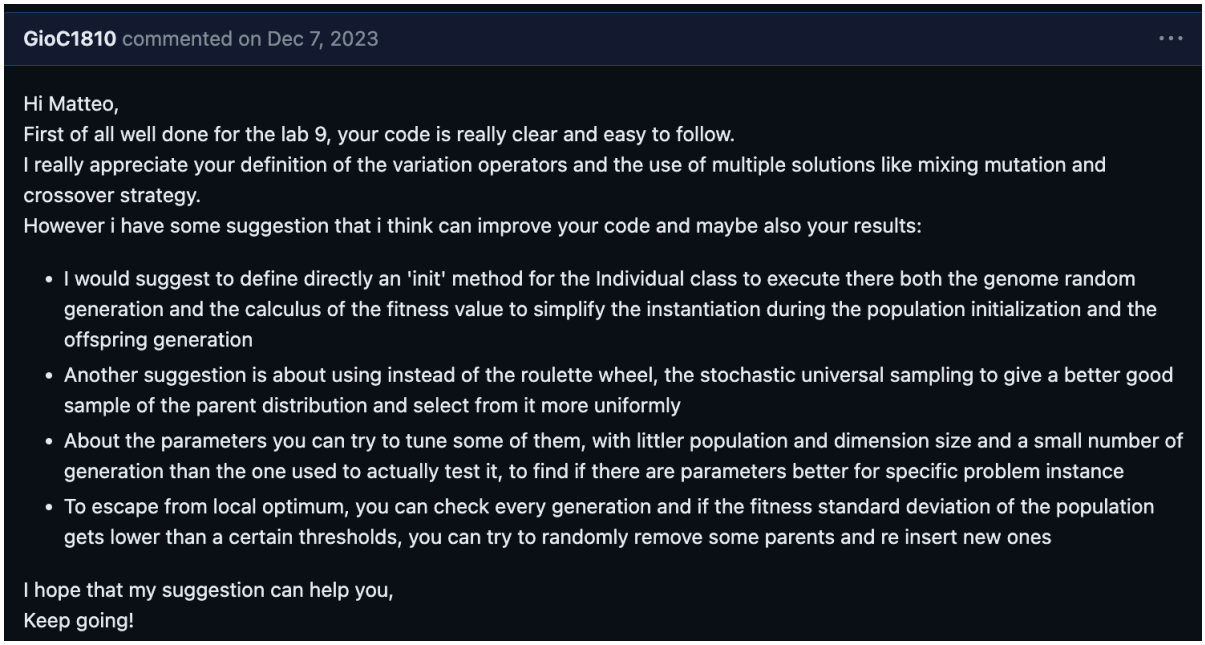
  - Cycle with standard deviation check

```
def launch_es_cycle_with_survival_selection(problem_instance: int, populations_number: int, n_loci: int, generations: int, offspring_size: int,
 tournament_size: int, selection_type: int, operator_agent: "Operators_agent", cross_operation: int, parent_selection_size: int):

    fitness_func = lab9_lib.make_problem(problem_instance)
    population = [Individual(fitness_func, n_loci) for _ in range(populations_number)]

    for gen in range(generations):

        offspring = offspring_generation(population, offspring_size, operator_agent, tournament_size, cross_operation)
        population = sorted(population+offspring if selection_type == 1 else offspring, key=lambda p: p.fitness, reverse=True)[:populations_number]
        if math.isclose(1, population[0].fitness):
            break
        if np.std([p.fitness for p in population]) < 0.005:
            best_individual = population[0]
            population.remove(population[0])
            population = stochastic_universal_sampling(population, parent_selection_size)
            population.append(best_individual)
            population.extend([Individual(fitness_func, n_loci) for _ in range(populations_number-len(population))])

    return population[0], fitness_func.calls
```

  - Islands model (implemented after the lab delivery)

```
def launch_islands_model(problem_instance: int, populations_number_per_islands: int, islands_number: int, migrants_number: int, n_loci: int,
 generations: int, offspring_size: int, tournament_size: int, selection_type: int, operator_agent: "Operators_agent", cross_operation: int,
 parent_selection_size: int, distance_function):

    fitness_func = lab9_lib.make_problem(problem_instance)
    islands_list = [Island(populations_number_per_islands, fitness_func, n_loci) for _ in range(islands_number)]

    for gen in range(generations):
        if (gen % 5 == 0):
            for _ in range(islands_number//2):
                index_1 = np.random.randint(0, islands_number)
                index_2 = np.random.randint(0, islands_number)
                while index_1 == index_2:
                    index_2 = np.random.randint(0, islands_number)
                island1, island2 = islands_list[index_1], islands_list[index_2]
                move_migrants(island1, island2, migrants_number, distance_function)
        for island in islands_list:
            offspring = offspring_generation_mixture(island.population, offspring_size, operator_agent, tournament_size, cross_operation)
            island.population = sorted(island.population+offspring if selection_type == 1 else offspring, key=lambda p: p.fitness, reverse=True)
             [:populations_number_per_islands]
            if math.isclose(1, island.population[0].fitness):
                break
            if np.std([p.fitness for p in island.population]) < 0.005:
                best_individual = island.population[0]
                island.population.remove(island.population[0])
                island.population = stochastic_universal_sampling(island.population, parent_selection_size)
                island.population.append(best_individual)
                island.population.extend([Individual(fitness_func, n_loci) for _ in range(populations_number_per_islands-len(island.population))])
        best = None
        for island in islands_list:
            best_of_island = island.take_best()
            if best is None or (best_of_island.fitness > best.fitness):
                best = best_of_island


    return best, fitness_func.calls
```

- I make the reviews for:
    - Matteo Celia s316607: https://github.com/Matteo-Celia/Computational-Intelligence-2023-2024.git



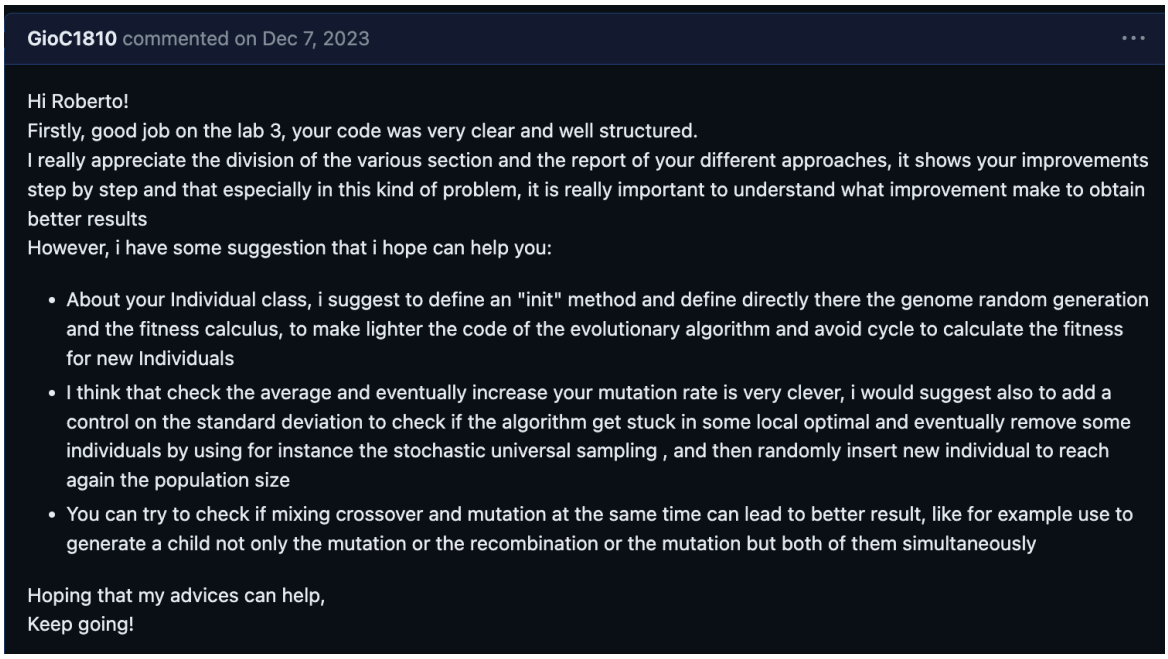
GioC1810 commented on Dec 7, 2023

Hi Matteo,
First of all well done for the lab 9, your code is really clear and easy to follow.
I really appreciate your definition of the variation operators and the use of multiple solutions like mixing mutation and crossover strategy.
However i have some suggestion that i think can improve your code and maybe also your results:

- I would suggest to define directly an 'init' method for the Individual class to execute there both the genome random generation and the calculus of the fitness value to simplify the instantiation during the population initialization and the offspring generation
- Another suggestion is about using instead of the roulette wheel, the stochastic universal sampling to give a better good sample of the parent distribution and select from it more uniformly
- About the parameters you can try to tune some of them, with littler population and dimension size and a small number of generation than the one used to actually test it, to find if there are parameters better for specific problem instance
- To escape from local optimum, you can check every generation and if the fitness standard deviation of the population gets lower than a certain thresholds, you can try to randomly remove some parents and re insert new ones

I hope that my suggestion can help you,
Keep going!

    - Roberto Pulvirenti s317704: https://github.com/ImBlurryF4c3/computationalIntelligence



GioC1810 commented on Dec 7, 2023

Hi Roberto!
Firstly, good job on the lab 3, your code was very clear and well structured.
I really appreciate the division of the various section and the report of your different approaches, it shows your improvements step by step and that especially in this kind of problem, it is really important to understand what improvement make to obtain better results
However, i have some suggestion that i hope can help you:

- About your Individual class, i suggest to define an "init" method and define directly there the genome random generation and the fitness calculus, to make lighter the code of the evolutionary algorithm and avoid cycle to calculate the fitness for new Individuals
- I think that check the average and eventually increase your mutation rate is very clever, i would suggest also to add a control on the standard deviation to check if the algorithm get stuck in some local optimal and eventually remove some individuals by using for instance the stochastic universal sampling , and then randomly insert new individual to reach again the population size
- You can try to check if mixing crossover and mutation at the same time can lead to better result, like for example use to generate a child not only the mutation or the recombination or the mutation but both of them simultaneously

Hoping that my advices can help,
Keep going!

Lab 4

- At first, I try to build a clear and convenient structure to model the tic tac toe game, so that it facilitates me in building the q agent strategy
- To actually build the q table strategy I take some inspiration by some colleagues and from book and online resources like DeepLizard YouTube channel and the book: Reinforcement Learning: an Introduction by Sutton
- At first I try to use a linear decrease of the exploration rate and I only used an epsilon greedy strategy, the results were not so bad but they were improvable
- So, i change the exploration rate decrease strategy to an exponential ones, I also tune a bit some parameter for the q table updating formula and I take the suggestion of Davide Vitabile for the minus sign in the formula, and that lead to better results
- Additinally, searching online I found and then implement other strategies to change the move choice and try a different approach for the exploration-exploitation trade off using as resource: Comparing Exploration Strategies for Q-learning in Random Stochastic Mazes from the University of Groningen
- I found that the UCB strategies obtains more victories against the random agent but it makes also less draws and loses more
- Here the move implementation with the 3 different strategies:
  - The first is the epsilon greedy policy
  - The second an upper bound confidence policy
  - The third a softmax or Boltzmann policy

```python
def move(self, state, possible_moves):
    converted_state = self.convert_state(state)
    if converted_state not in self.q_table:
            self.q_table[converted_state] = np.zeros((9,))
    if self.exploration_strategy == 0:
        if random() < self.exploration_rate:
            return tuple(choice(possible_moves))
        else:
            possible_moves = [self.convert_action(action) for action in possible_moves]
            possible_values = [self.q_table[converted_state][action]  for action in possible_moves]
            max_value = max(possible_values)
            best_moves = [action for action, value in zip(possible_moves, possible_values) if value == max_value]
            move = choice(best_moves)
            return move // 3, move % 3
    elif self.exploration_strategy == 1:
        if random() < self.exploration_rate:
            return tuple(choice(possible_moves))
        else:
            possible_moves = [self.convert_action(action) for action in possible_moves]
            ucb_values = [self.q_table[converted_state][action] + np.sqrt(2 * np.log(len(possible_moves)) /
                            max(1, np.sum(self.q_table[converted_state][action])))
                          for action in possible_moves]
            move = possible_moves[np.argmax(ucb_values)]
            return move // 3, move % 3
    else:
        possible_moves = [self.convert_action(action) for action in possible_moves]
        possible_values = [self.q_table[converted_state][action]  for action in possible_moves]
        max_value = np.max(possible_values)
        scaled_values = [val - max_value for val in possible_values]
        exp_values = np.exp(np.array(scaled_values) / self.exploration_rate)
        boltzmann_probs = exp_values / np.sum(exp_values)
        chosen_action_index = np.random.choice(len(possible_moves), p=boltzmann_probs)
        move = possible_moves[chosen_action_index]
        return move // 3, move % 3
```

- In the end I also trained the agent as second player to increase its performance playing as second player
- The final result are this:

```python
agent_greedy = Q_Agent(learning_rate=0.2,
                       discount_rate=0.8,
                       exploration_rate= 1,
                       min_exploration_rate=0.01,
                       exploration_decay= 3e-6,
                       opponent=RandomAgent(1),
                       exploration_strategy=0)
agent_greedy.train(500000)
print(f"Exploration rate: {agent_greedy.exploration_rate}")
N_MATCHES = 10000
agent_greedy.exploration_rate = 0
test_agent(agent_greedy, RandomAgent(1), N_MATCHES)
```

Executed at 2023.12.24 16:42:15 in 56s 940ms

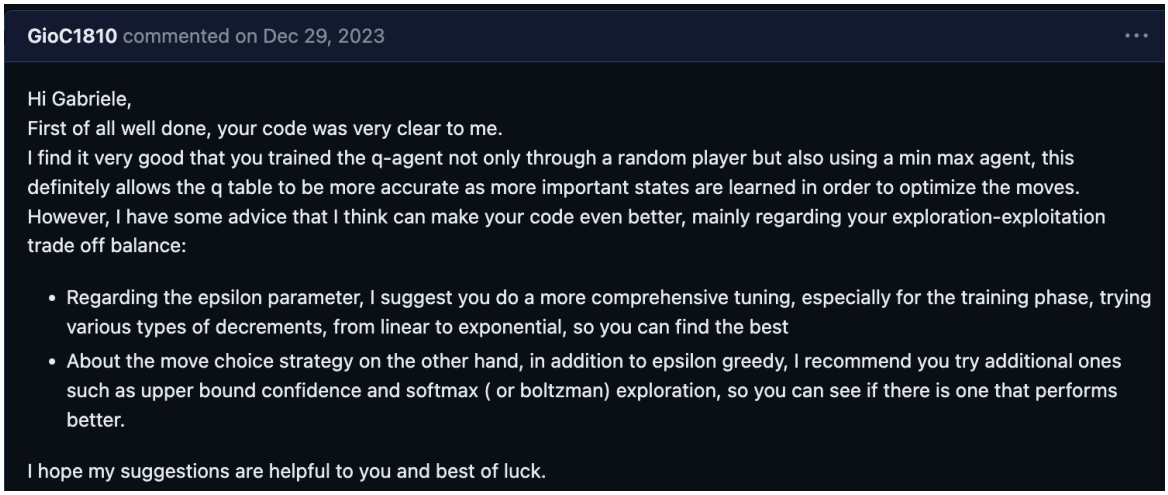
```
Exploration rate: 0.22313082953991437
Wins: 7554, draws: 1307 over 10000
Explored states: 5162

(7554, 1307)
```

```python
agent_ucb = Q_Agent(learning_rate=0.1,
                    discount_rate=0.99,
                    exploration_rate= 1,
                    min_exploration_rate=0.01,
                    exploration_decay= 3e-6,
                    opponent=RandomAgent(0),
                    exploration_strategy=1)
agent_ucb.train(2000000)
N_MATCHES = 10000
agent_ucb.exploration_rate = 0
test_agent(agent_ucb, RandomAgent(1), N_MATCHES)
```

Executed at 2023.12.24 16:07:35 in 5m 36s 841ms

```
Wins: 8383, draws: 547 over 10000
Explored states: 5162

(8383, 547)
```

- I make the reviews for:
    - Gabriele Quaranta s318944: https://github.com/gabriquaranta/computational-intelligence



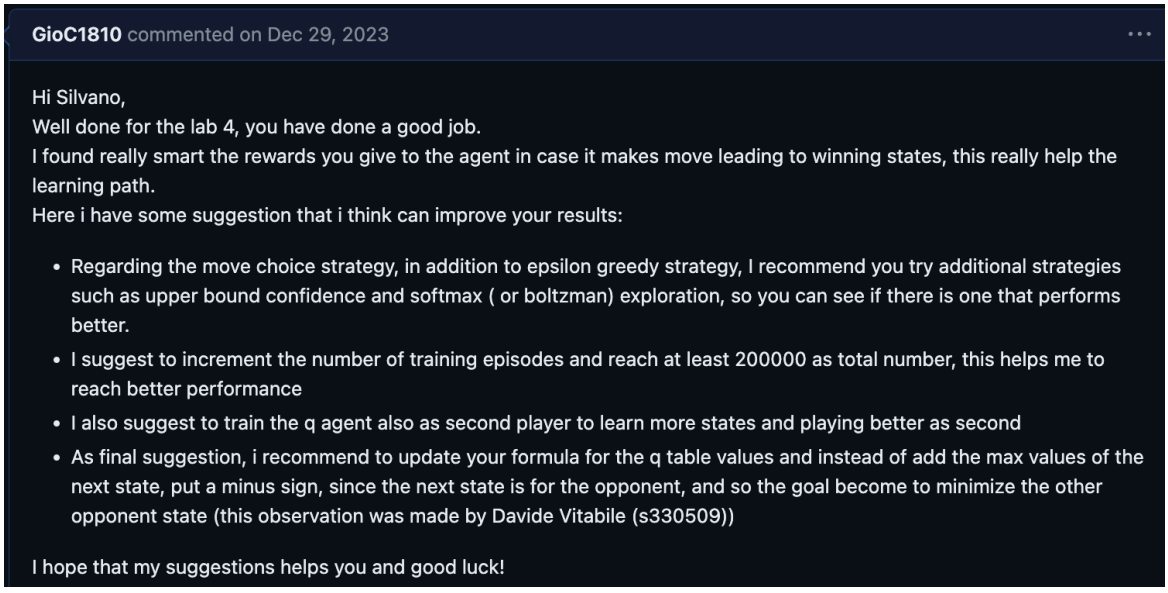
GioC1810 commented on Dec 29, 2023

Hi Gabriele,
First of all well done, your code was very clear to me.
I find it very good that you trained the q-agent not only through a random player but also using a min max agent, this definitely allows the q table to be more accurate as more important states are learned in order to optimize the moves.
However, I have some advice that I think can make your code even better, mainly regarding your exploration-exploitation trade off balance:

- Regarding the epsilon parameter, I suggest you do a more comprehensive tuning, especially for the training phase, trying various types of decrements, from linear to exponential, so you can find the best
- About the move choice strategy on the other hand, in addition to epsilon greedy, I recommend you try additional ones such as upper bound confidence and softmax (or boltzman) exploration, so you can see if there is one that performs better.

I hope my suggestions are helpful to you and best of luck.

    - Silvano Silvano s320168: https://github.com/s320168/ComputationalIntelligence



GioC1810 commented on Dec 29, 2023

Hi Silvano,
Well done for the lab 4, you have done a good job.
I found really smart the rewards you give to the agent in case it makes move leading to winning states, this really help the learning path.
Here i have some suggestion that i think can improve your results:

- Regarding the move choice strategy, in addition to epsilon greedy strategy, I recommend you try additional strategies such as upper bound confidence and softmax (or boltzman) exploration, so you can see if there is one that performs better.
- I suggest to increment the number of training episodes and reach at least 200000 as total number, this helps me to reach better performance
- I also suggest to train the q agent also as second player to learn more states and playing better as second
- As final suggestion, i recommend to update your formula for the q table values and instead of add the max values of the next state, put a minus sign, since the next state is for the opponent, and so the goal become to minimize the other opponent state (this observation was made by Davide Vitabile (s330509))

I hope that my suggestions helps you and good luck!

- After the delivery of the lab, while working on Quixo I tried to change the reward strategies of my agent
  - Instead of assign to my agent a reward of 1 each move, I track every states and move performed by the agent and at the end of the game I backpropagate the reward to all the states traversed
  - I also notice that instead of applying the same reward value to each state, give a proportional reward value depending on how much the game is closed to the end lead to better results and that improve the performance of my agent

```
agent_boltzmann = Q_Agent(learning_rate=0.1,
                  discount_rate=0.99,
                  exploration_rate= 1,
                  min_exploration_rate=0.01,
                  exploration_decay= 3e-6,
                  opponent=RandomAgent(0),
                  exploration_strategy=2)
agent_boltzmann.train(1000000)
Executed at 2024.02.07 17:09:18 in 2m 46s 421ms

N_MATCHES = 10000
agent_boltzmann.exploration_rate = 0.00001
test_agent(agent_boltzmann, RandomAgent(1), N_MATCHES)
Executed at 2024.02.07 17:10:31 in 1s 509ms

 Wins: 7587, draws: 1242 over 10000
 Explored states: 5162

 (7587, 1242)
```

```
def train_backprop_incremental(self, n_episodes, first_Player=True):
    game = Tic_Tac_Toe()
    self.player_number = 0 if first_Player else 1
    self.opponent.player_number = 1-self.player_number
    players = [self, self.opponent]
    turn = 0 if first_Player else 1
    for episode in range(n_episodes):
        states_action_traversed = []
        while game.state == -1:
            possible_moves = game.possible_moves()
            actual_state = self.convert_state(game.board)
            if game.actual_player == self.player_number:
                action = players[turn].move(game.board, possible_moves)
                states_action_traversed.append((actual_state, action))
                _, game_state = game.move(action, players[turn].player_number)
                next_state = self.convert_state(game.board)
                #self.update_q_table(actual_state, action, reward, next_state)
            else:
                game.move(players[turn].move(game.board, possible_moves), players[turn].player_number)
            turn = 1-turn
        turn = 0 if first_Player else 1
        game_reward = self.get_game_reward(game_state)
        self.update_q_table(actual_state, action, game_reward, next_state)
        reward_step_decrement = game_reward / len(states_action_traversed)
        for state, action in states_action_traversed[::-1]:
            self.update_q_table(state, action, game_reward, next_state)
            game_reward -= reward_step_decrement if game_reward > 0 else -reward_step_decrement
        game.reset()
        self.exploration_rate = np.clip(
            np.exp(-self.exploration_decay * episode), self.min_exploration_rate, 1
        )
```

- The result obtained are significantly better as can be seen

```
agent_greedy = Q_Agent(learning_rate=0.2,
                  discount_rate=0.8,
                  exploration_rate= 1,
                  min_exploration_rate=0.01,
                  exploration_decay= 3e-6,
                  opponent=RandomAgent(1),
                  exploration_strategy=2)
agent_greedy.train_backprop_incremental(1000000)
N_MATCHES = 10000
agent_greedy.exploration_rate = 0.00001
test_agent(agent_greedy, RandomAgent(1), N_MATCHES)
```

Executed at 2024.01.24 14:53:28 in 2m 35s 777ms

```
Wins: 9955, draws: 45 over 10000
Explored states: 3925

(9955, 45)
```

Quixo

- The first agent I tried to implement implemented a **vanilla q learning algorithm** similar to the ones implemented in the lab 10 with the back propagation
- The training of it last a lot of time and the resulting table was too large, around 19GB, in addition to that the performance were very bad since the winning rate was just above 50%
- To improve both space and performance I tried to implement some symmetries to reduce the possible states explored and memorize in the table just the canonical form of every state
- Basically, the agent takes the current board state, convert it into its canonical form and then perform the move based on the epsilon greedy police
- After performing the move, I have to convert the action applied to the canonical form to an action equivalent but on the original state so I implemented

```python
@staticmethod
def apply_rotation(state_np, rotations):
    # Rotate the state by the specified number of rotations (clockwise)
    return np.rot90(state_np.reshape((5, 5)), k=rotations).flatten()

1 usage  Giorgio Cacopardi
@staticmethod
def apply_reflection(state_np, axis):
    # Reflect the state along the specified axis (0 for vertical, 1 for horizontal)
    return np.flip(state_np.reshape((5, 5)), axis=axis).flatten()

1 usage  Giorgio Cacopardi *
@staticmethod
def apply_symmetry(state_str, rotations=0, reflection_axis=None):
    state_np = np.array([int(char) for char in state_str])

    state_np = Utilities.apply_rotation(state_np, rotations)

    if reflection_axis is not None:
        state_np = Utilities.apply_reflection(state_np, axis=reflection_axis)

    return ''.join(map(str, state_np))

2 usages  Giorgio Cacopardi
@staticmethod
def convert_action_to_triplet(action):
    return ((action // 20) % 5, (action // 4) % 5), Move(action % 4)

4 usages  Giorgio Cacopardi
@staticmethod
def convert_action_to_scalar(action):
    return action[0][0] * 20 + action[0][1] * 4 + action[1].value

6 usages  Giorgio Cacopardi
@staticmethod
def get_canonical_state(state_str):
    state_str = Utilities.convert_state(state_str)
    state_np = np.array([int(char) for char in state_str])
    state_str = ''.join(map(str, state_np))
    canonical_state = state_str
    transformations = (0, None)

    for rotations, reflection_axis in TRANSFORMATIONS:
        sym_state = Utilities.apply_symmetry(state_np, rotations, reflection_axis)
        if sym_state < canonical_state:
            canonical_state = sym_state
            transformations = (rotations, reflection_axis)

    return canonical_state, transformations
```

some methods to apply in the inverse order the transformations previously made to the state

- In the end this changes does not lead to any remarkable improvement since the number of states and possible actions is too large and the only way to make a working algorithm is to use deep q learning to approximate the state space

```python
@staticmethod
def get_reverse_action(action, transformations):
    canonical_action = action
    rotations, reflection_axis = transformations

    if reflection_axis is not None:
        canonical_action = Utilities.reverse_reflection(canonical_action, reflection_axis)

    for _ in range(rotations % 4):
        canonical_action = Utilities.reverse_rotation(canonical_action)

    return canonical_action

1 usage  Giorgio Cacopardi *
@staticmethod
def reverse_reflection(action, axis):
    if axis == 1:
        from_pos = (action[0][0], 4 - action[0][1])
        if action[1] in [Move.TOP, Move.BOTTOM]:
            return (from_pos, action[1])
        else:
            if action[1] == Move.LEFT:
                move = Move.RIGHT
            else:
                move = Move.LEFT
            return (from_pos, move)
    elif axis == 0:
        from_pos = (4 - action[0][0], action[0][1])
        if action[1] in [Move.LEFT, Move.RIGHT]:
            return (from_pos, action[1])
        else:
            if action[1] == Move.TOP:
                move = Move.BOTTOM
            else:
                move = Move.TOP
            return (from_pos, move)

1 usage  Giorgio Cacopardi
@staticmethod
def reverse_rotation(action):
    return ((action[0][1], 4 - action[0][0]), action[1].clockwise())
```

- The principal part of the agent are the following

```python
def make_move_train(self, game) -> tuple[tuple[int, int], Move]:


    if np.random.rand() < self.exploration_rate:
        action = random.choice(Utilities.generate_valid_moves(game, self.player_number))
    else:
        canonical_state, transformations = Utilities.get_canonical_state(game.get_board())
        game_copy = deepcopy(game)
        game_copy._board = Utilities.convert_string_to_array(canonical_state)

        q_values = self.q_table[canonical_state]
        valid_moves = [Utilities.convert_action_to_scalar(action) for action in
                       Utilities.generate_valid_moves(game_copy, self.player_number)]
        valid_q_values = [q_values[action] for action in valid_moves]
        max_q_value = max(valid_q_values)
        max_index_es = [action for action, q_value in zip(valid_moves, valid_q_values) if q_value == max_q_value]
        action = np.random.choice(max_index_es)
        action = Utilities.convert_action_to_triplet(action)
        action = Utilities.get_reverse_action(action, transformations)
        if not deepcopy(game)._Game__move(action[0], action[1], self.player_number):
            action = random.choice(Utilities.generate_valid_moves(game, self.player_number))
            self.update_q_table(canonical_state, Utilities.convert_action_to_scalar(action), -50,  next_state: None)
    return action

2 usages  Giorgio Cacopardi
def make_move(self, game) -> tuple[tuple[int, int], Move]:

    canonical_state, transformations = Utilities.get_canonical_state(game.get_board())
    game_copy = deepcopy(game)
    game_copy._board = Utilities.convert_string_to_array(canonical_state)

    q_values = self.q_table[canonical_state]
    valid_moves = [Utilities.convert_action_to_scalar(action) for action in
                   Utilities.generate_valid_moves(game_copy, self.player_number)]
    valid_q_values = [q_values[action] for action in valid_moves]
    max_q_value = max(valid_q_values)
    max_index_es = [action for action, q_value in zip(valid_moves, valid_q_values) if q_value == max_q_value]
    action = np.random.choice(max_index_es)
    action = Utilities.convert_action_to_triplet(action)
    action = Utilities.get_reverse_action(action, transformations)
    if not deepcopy(game)._Game__move(action[0], action[1], self.player_number):
        action = random.choice(Utilities.generate_valid_moves(game, self.player_number))
    return action
```

```python
def update_q_table(self, state, action, reward, next_state=None):
    if next_state is not None:
        self.q_table[state][action] = ((1 - self.learning_rate) * self.q_table[state][action] +
                                       self.learning_rate * (reward + self.discount_rate * (
                    -np.max(self.q_table[next_state]))))
    else:
        self.q_table[state][action] = reward

1 usage  Giorgio Cacopardi
def execute_move(self, game: Game, states_action_traversed):

    canonical_state, transformations = Utilities.get_canonical_state(game.get_board())
    action = self.make_move_train(game)
    game._Game__move(action[0], action[1], self.player_number)
    next_state = Utilities.get_canonical_state(game.get_board())[0]
    states_action_traversed.append((canonical_state, Utilities.convert_action_to_scalar(action), next_state))
```

```python
Giorgio Cacopardi
def train(self, n_episodes, opponent, first_Player=True):
    game = Game()
    self.player_number = 1 if first_Player else 2
    opponent.player_number = 2 if first_Player else 1
    players = [self, opponent]
    turn = 0 if first_Player else 1
    episode_count = 0
    for episode in range(n_episodes):
        states_action_traversed = []
        counter = 0
        while game.check_winner() == -1 and counter < 200:
            if turn == self.player_number-1:
                self.execute_move(game, states_action_traversed)
            else:
                ok = False
                while not ok:
                    move = players[turn].make_move(game)
                    ok = game._Game__move(move[0], move[1], players[turn].player_number)
            turn = 1 - turn
            counter += 1
        turn = 0 if first_Player else 1
        game_reward = self.get_game_reward(game.check_winner())
        reward_decrementation = game_reward / len(states_action_traversed)
        for state, action, next_state in states_action_traversed[::-1]:
            self.update_q_table(state, action, game_reward, next_state)
            game_reward -= reward_decrementation if game_reward > 0 else -reward_decrementation
        game = Game()
        self.exploration_rate = np.clip(
            np.exp(-self.exploration_decay * episode), self.min_exploration_rate,  a_max: 1)
        episode_count += 1
        if episode_count % 100 == 0:
            print(f"Episode: {episode_count} with exploration rate: {self.exploration_rate}")
```

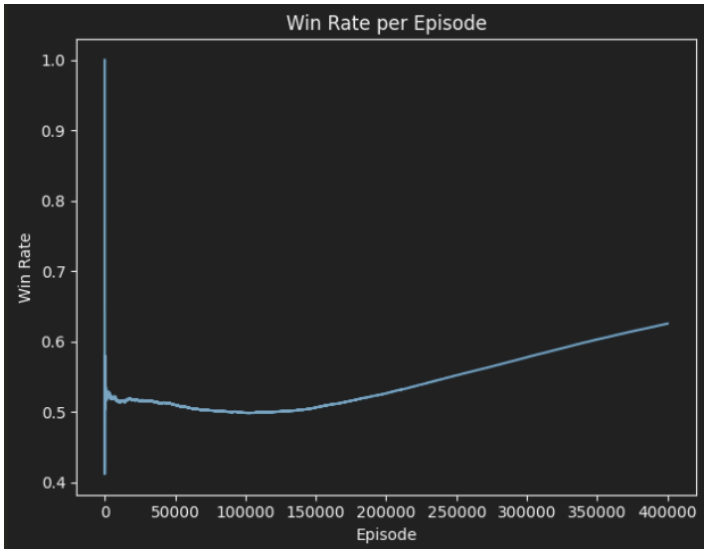
- The next agent using reinforcement learning approach is the **monte carlo agent** that employs a temporal difference method
- This agent handles significantly better the huge number of spaces and also the application of symmetries is more manageable compared to the q learning agent
- About the algorithm strategy I employed an epsilon greedy approach and also some symmetries to obtain the canonical form of the states
- The agent from the actual state calculate all the possible next transitions from it and take the states with the best expected rewards according to the state value table
- To further reduce the size of the table, that in the beginning was about 1.9GB I change the missing method of the default dict used to implement the state value table to avoid to insert useless states where the key is not present so that the table memorize just the necessary state, and obtaining a complessive size of 53.6MB
- I also employed job lib library to compress more the table compared to the standard pickle library
- The principal method are the following

```
def make_move(self, game) -> tuple[tuple[tuple[int, int], Move], str]:

    if np.random.rand() < self.exploration_rate:
        action = random.choice(Utilities.generate_valid_moves(game, self.player_number))
        game_copy = deepcopy(game)
        game_copy._Game__move(action[0], action[1], self.player_number)
        next_state = Utilities.get_canonical_state(game_copy.get_board())[0]
    else:
        canonical_transition = Utilities.generate_canonical_transitions(game, self.player_number)
        state_chosen, action = max(canonical_transition, key=lambda t: self.state_value[t[0]])
        next_state = state_chosen

    return action, next_state
def update_q_table(self, game_reward):
    reward = game_reward
    for state in reversed(self.states_traversed):
        self.state_value[state] = self.state_value[state] + self.learning_rate * (
                self.discount_rate * reward - self.state_value[state])
        reward = self.state_value[state]
    self.states_traversed = []
```

- About the training, it lasts for about 5h and I used an exponential decay that best suits the chosen training episodes number

```
self.exploration_decay = -np.log(self.min_exploration_rate) / n_episodes
```

- The results about the winning rate are showed in this plot and as can be seen, the agents learn constantly as the number of episodes increase



- The final winning rate with 400000 episodes of training is about 84% both for playing as first or second player

- The last agent created was the **minimax agent** that basically employ a minimal algorithm with alpha beta pruning since the branching factor of Quito is huge and the time to make a move drastically increase
- I chose to adopt a depth of 2 since it is the best trade between time for make a move and quality of the move, since a depth of 3 does not improve by a lot the win rate but provoke a drastically higher time to make a move
- The heuristic adopted for evaluating non final states was the following that works by giving a score of 1 to every row, column or diagonal with 4 cube of the current player id, and 0.5 to every line with 3 cube
- The performance are better compared to the monte carlo agent since it reaches almost 100% of winning rate (98%)

```python
def heuristic(self, game, player_id):

    if game.check_winner() == 1:
        return 10
    elif game.check_winner() == 2:
        return -10
    else:
        score = 0
        board = game.get_board()
        # Check rows
        for row in board:
            player_cells = np.sum(row == player_id)
            if player_cells == 4:
                score += 1
            if player_cells == 3:
                score += 0.5

        # Check columns
        for col in board.T:
            player_cells = np.sum(col == player_id)
            if player_cells == 4:
                score += 1
            if player_cells == 3:
                score += 0.5

        # Check diagonals
        diag1 = board.diagonal()
        diag2 = np.fliplr(board).diagonal()
        for diag in [diag1, diag2]:
            player_cells = np.sum(diag == player_id)
            if player_cells == 4:
                score += 1
            if player_cells == 3:
                score += 0.5

        return score
```