Computational Intelligence

Report of the year

Berardo Nicholas s319439

Politecnico di Torino

Chapter 1

8-puzzle

The code doesn't work beacuse I get this error:

```
ValueError: The truth value of an array with more than one element is ambiguous.
Use a.any() or a.all()

```

The source of the error is given by the put() method of the PriorityQueue. At some point it computes a $\leq$ operation on the elements of the PriorityQueue and this error is given. I don't know why but for the first 3 iteration it works then stops...

By the way this is the main code:

```
queue = PriorityQueue()
initial_state = (np.array([2,5,7,1,0,8,9,3,4]),np.array([5,1,8,3]))
queue.put((distance_check(initial_state[0]),initial_state))

_,state = queue.get()
current_state,state_to_move = state
while not goal_check(current_state):
    idx_hole = np.where(current_state==0)[0].item()
    for i in state_to_move:
        idx_to_change = np.where(current_state==i)[0].item()
        tmp = current_state.copy()
        tmp[idx_hole] = i
        tmp[idx_to_change] = 0
        tmp_move = []
        if idx_to_change-3>=0:
            tmp_move.append(tmp[idx_to_change-3])

        if idx_to_change+3<3*3:
            tmp_move.append(tmp[idx_to_change+3])

        if idx_to_change > 0 and idx_to_change>3*np.floor((idx_to_change)/3):
            tmp_move.append(tmp[idx_to_change-1])

        if idx_to_change<8 and idx_to_change+1<3+3*np.floor((idx_to_change)/3):
            tmp_move.append(tmp[idx_to_change+1])

        new_state = (np.array(tmp),np.array(tmp_move))
        print(distance_check(new_state[0]))

        queue.put((distance_check(new_state[0]),new_state))

    _,state = queue.get()
    current_state,state_to_move = state
```

Chapter 2

A*

With A* i adoped an h that is pretty easy to understand. Let's assume we are in this case:

- $PROBLEMSIZE = 20$
- $NUMSETS = 10$

This h computes the mean distance from the goal for the uncovered sets.

```
    def h(state):
    return np.ceil((sum(sum(sets[s] for s in state[1]))) / (PROBLEM_SIZE-len(state
    [0])))
```

Chapter 3

Halloween

I tried to implement the Halloween Challenge with four single state algorithm:

- Hill Climbing
- Simulated Annealing
- Tabu Search
- Iterated Local Search

This is the definition of my goal check:

```
    def goal_check(state):
        cost = sum(state)
        is_valid = set(np.concatenate(x.rows[state])) == set(i for i in range(
    num_points))
        return is_valid,-cost
```

Meanwhile this is the tweak function

```
    def tweak(state):
        new_state = state.copy()
        for i in range(int(30*num_points/100)):
            index = randint(0, num_points - 1)
            new_state[index] = not new_state[index]
        return new_state
```

3.1 Hill Climbing

```
    state = initiale_state
    for step in range(num_points):
        new_state = tweak(state)
        if goal_check(state)<goal_check(new_state):
            state=new_state
```

If the new state is better then the old state, update it. Pretty easy and fast. On a 100 point example it achive a cost of 32.

3.2 Simulated Annealing

```
state = initiale_state
t=num_points
for step in range(num_points):
    new_state = tweak(state)
    oldStateCost = goal_check(state)
    newStateCost = goal_check(new_state)
    p=np.exp(-((oldStateCost[1]-newStateCost[1])/t)) #generate the probability
    t=t-1
    if oldStateCost>=newStateCost and random() < p : #accept a worse solution
with a prob p!!
        state=new_state
    else:
        if oldStateCost<newStateCost:
            state=new_state
```

In this algorithm we accept a state that is worse then the previous state with a probability p. The main idea is that when t (also called temperature) is high, we are exploring, so we accept worse solutions. When t goes closer to 0, the probability will go to 0, so we won't accept worse solution.

3.3 Tabu Search

```
state = initiale_state
tabu_list = []
for step in range(num_points):
    new_sol_list = [tweak(state) for _ in range(int(5*num_points/100))]
    candidates = [sol for sol in new_sol_list if is_valid(sol) and sol not in
tabu_list]
    if not candidates:
        continue
    for sol in candidates:          #take the best option from the generated
ones
        tabu_list.append(sol)
        if goal_check(sol)>goal_check(state):
            state = sol
```

In this algorithm we have a tabuList where we save all the solution we have already checked. Then in the future we will accept in the candidate solution only the ones that aren't in the tabuList, so the main idea is to avoid looking for already checked solutions!! Also at each iteration we generate 5/100 of numPoints random state.

3.4 Iterated Local Search

```
def hill_climbing(initiale_state):
state = initiale_state
for step in range(num_points):
    new_state = tweak(state)
    if goal_check(state)<goal_check(new_state):
        state=new_state
return state
```

Define this function that is the Hill Climbing

```
    N_Restart = 10
    state = initiale_state
    for i in range(N_Restart):
        solution = hill_climbing(state)
        state = solution
```

The idea is to restart the algorithm where the previous ”iteration” stopped, with the previous optimal state.

Chapter 4

Lab2-ES NimSum

First of all I want to say that I cooperated with Riccardo Cardona for this laboratory.

The main idea of our solution is to use some kind of probability to choose the best move.

```
    def make_move(self, state) -> Nimply:
        #(row,obj)
        moves = [
            (row, min(state.rows[row], max(1, int(self.individual[row] * state.rows[
    row]))))
            for row in range(len(state.rows)) if state.rows[row] > 0]

        # Select the move with the highest preference, the highest preference is the
     one with the highest genome value.
        # Self.individual[x[0]] indicates the probability associated with the row of
     the game board x[0] that we want to modify
        chosen_move = max(moves, key=lambda x: self.individual[x[0]])

        # Return the move
        return Nimply(*chosen_move) #the * is used for unpacking, chosen_move is an
    array!!! We want only the element inside
```

The fuction above pick the best move based on the probability of the individual in the population. On individual is composed of N porbabilities, where N is the number of rows in the nim game. The function computes for each row the possible move that is (row,n_object), where n_object will be a number between 1 and the actual number of objects in that row. Once we have the moves we choose the one which row correspond to the highest probability of the individual.
Example: $n = 5$, *individual* = [0.1, 0.5, 0.4, 0.6, 0.9], *moves* = [(0,1), (1,1), (2,1), (3,4), (4,6)]
The *individual* tells us that the move (4,6) is the one with higest prob!!!

```
    def reproduce(selected_agents, mutation_rate, mutation_probability):
        new_population = []

        while len(new_population) < len(selected_agents):
            if random.random() < mutation_probability:
                parent1 = random.sample(selected_agents, 1)[0]
                child_genome = mutation(parent1.individual, mutation_rate)
            else:
                parent1, parent2 = random.sample(selected_agents, 2)
                child_genome = crossover(parent1.individual, parent2.individual)
            new_population.append(NimAgent(child_genome))
        return new_population

    def mutation(genome, mutation_rate, mu=0, sigma=1):
        for prob in range(len(genome)):
            if random.random() < mutation_rate:
                mutation = random.gauss(mu,sigma)
                genome[prob] = min(max(genome[prob] + mutation, 0), 1)
        return genome

    def crossover(genome1, genome2):
        child_genome = [g1 if random.random() > 0.5 else g2 for g1, g2 in zip(
    genome1, genome2)]
        return child_genome

    def replacement(population, new_population, fitness_scores):
        sorted_population = sorted(zip(population, fitness_scores), key=lambda x: x
    [1], reverse=True)
        survivors = sorted_population[:len(population) - len(new_population)]
        return [agent for agent, score in survivors] + new_population
```

This methods are needed for the reproduction of the population. Classic mutation and xover function. The replacement method orders the current population based on fitness and then substitute the last X elements with the new population that is given in output from the reproduce().

```
    def evaluate_population(population : [NimAgent], nim: Nim, num_games: int):
        wins = []
        for individual in population:
            strategy = (optimal,individual.make_move)
            win = 0
            for _ in range(num_games):
                nim.reset()
                player = 0
                while nim:
                    if player == 1:
                        ply = strategy[player](nim)
                    else:
                        ply = strategy[player](nim)
                    nim.nimming(ply)
                    player = 1 - player
                if player == 1:
                    win+=1
            wins.append(win)
        return wins

    def select_agents(population, fitness_scores):
        selected = []
        while len(selected) < len(population) // 2:
            participant = random.sample(list(zip(population, fitness_scores)), 2)
            winner = max(participant, key=lambda x: x[1])
            selected.append(winner[0])
        return selected
```

The evaluate_population method is our fitness function, it returns the number of win of the current population agains the optimal strategy (this could be exapanded to other strategies).

The select_agents method is the one that select the parents from the current population for the reproduction.

```
def evolutionary_strategy(nim, generations, population_size, initial_mutation_rate, wins_goal, num_games, mutation_probability):
    population = initialize_population(population_size, len(nim.rows))
    best_individual = None
    best_fitness = -1
    mutation_rate = initial_mutation_rate

    for generation in range(generations):
        fitness_scores = evaluate_population(population, nim, num_games)

        # The best score is halved to report the number of wins, since each win is worth double points in the scoring system.
        print(f"Generation {generation}: Best score {max(fitness_scores)} wins")

        # Check for termination condition (e.g., a perfect score)
        if max(fitness_scores) >= wins_goal:
            print("Stopping early, reached perfect score!")
            break

        # Selection
        selected_agents = select_agents(population, fitness_scores)

        # Reproduction
        new_population = reproduce(selected_agents, mutation_rate, mutation_probability)

        # Replacement
        population = replacement(population, new_population, fitness_scores)

        # Check if the new best individual is found
        max_fitness = max(fitness_scores)
        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_individual_index = fitness_scores.index(max_fitness)
            best_individual = population[best_individual_index]

    return population, best_individual
```

Finally, this is my loop through the generation.