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]))

```

```
30          queue.put((distance_check(new_state[0]),new_state))
31
32      _,state = queue.get()
33      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.

```
1     def h(state):
2     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:

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

Meanwhile this is the tweak function

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

3.1 Hill Climbing

```
1     state = initiale_state
2     for step in range(num_points):
3         new_state = tweak(state)
4         if goal_check(state)<goal_check(new_state):
5             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.