Report on Final Project Quarto

(Laboratories Included)

by Milad Zakhireh

student ID: s300708

link to repository:

labs: https://github.com/milad818/computational_intelligence_2022

final project: https://github.com/milad818/CI2022-23-Quarto-Final-Project

Laboratories

Laboratory assignments are done in group with my dear friend Masoud Karimi.

Lab 1

Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, \ldots, L_n)$, determine, if possible, $S = (L_{s_0}, L_{s_1}, L_{s_2}, \ldots, L_{s_n})$ such that each number between 0 and $N-1$ appears in at least one list

$$\forall n \in [0, N-1]\ \exists i : n \in L_{s_i}$$

and that the total numbers of elements in all L_{s_i} is minimum.

Instructions

- Create the directory `lab1` inside the course repo (the one you registered with Andrea)
- Put a `README.md` and your solution (all the files, code and auxiliary data if needed)
- Use `problem` to generate the problems with different N
- In the `README.md`, report the the total numbers of elements in L_{s_i} for problem with $N \in [5, 10, 20, 100, 500, 1000]$ and the total number on $nodes$ visited during the search. Use `seed=42`.
- Use `GitHub Issues` to peer review others' lab

Group Members: Milad Zakhireh, Masoud Karimi

__init__.py

```
Import
numpy
        import problem
        import state as S
        import property as P
        import search as SRCH



        N = 5
        seed = 42
        goal = set(range(N))
        initial = ([])
        parentState = dict()
        stateCost = dict()

        props = P.Properties(N, seed, goal, initial)
```

```
problem = problem.makeProblem(props.N, props.seed)
# problem = sorted(problem, key=lambda l: len(l))

initialState = S.State(props.initial)
goalState = S.State(props.goal)
# initialState = S.State(numpy.array([1,2,34,3,5]))

def h(newState : S , currentState : S ):
    # return (numpy.sum((currentState != goalState) &
(numpy.array(list(currentState.contain())))))).size
    # return numpy.sum( ( currentState != goalState ) &
(numpy.array(list(currentState.contain()))))
    # return len(currentState.contain())
    #
    _cS = currentState.copyData()
    _nS = newState.copyData()
    _cS = set(_cS)
    _nS = set(_nS)
    # _cS |= set(_nS)
    _cS &= set(_nS)
    return len (_cS)

path = SRCH.doesSearch(initialState, goalState, parentState, stateCost, problem,
                       unitCost= lambda s:1,
                       priorityFunction=lambda nS,cS : h(nS , cS) )
W = 0
for p in path:
    W += len(p)

print (f"path = {path}")
print(f"W = {W}")
```

priorityQueue.py

```
import
heapq

class PriorityQueue:
    """A basic Priority Queue with simple performance optimizations"""
```

```
    def __init__(self):
        self._data_heap = list()
        self._data_set = set()

    def __bool__(self):
        return bool(self._data_set)

    def __contains__(self, item):
        return item in self._data_set

    def push(self, item, p=None):
        assert item not in self, f"Duplicated element"
        if p is None:
            p = len(self._data_set)
        self._data_set.add(item)
        heapq.heappush(self._data_heap, (p, item))

    def pop(self):
        p, item = heapq.heappop(self._data_heap)
        self._data_set.remove(item)
        return item
```

problem.py

```
import
random
import numpy

def makeProblem(N, seed=None):
    random.seed(seed)
    return [ numpy.array(list(set(random.randint(0, N - 1) for n in
range(random.randint(N // 5, N // 2))))) for n in range(random.randint(N, N * 5))
]
```

property.py

```
class
Properties:
                def __init__(self, numberOfElements, seed, finalGoal, initialList):
                    self.N = numberOfElements
                    self.seed = seed
                    self.goal = list(finalGoal)
                    self.initial = list(initialList)
```

search.py

```
import
logging
          import priorityQueue as PQ
          import state
          import problem as PROB
          import numpy

          def goalStateCheck(stateToBeChecked : state, goal : state):
              return set(stateToBeChecked.contain()) == set(goal.contain())


          def resultOfAction(currentState : state, action: numpy.ndarray):
              # temp = currentState.contain()
              # temp.append(action)
              # temp = tuple(temp)
              # return temp

              #
              cS = set(currentState.copyData())
              cS |= set(action)
              return state.State(cS)



          def doesSearch ( initialStat: state,
                          goal : state,
                          parentState : dict,
                          stateCost: dict,
                          problem : PROB,
                          unitCost : callable,
```

```
                priorityFunction : callable
                 ):
    # print(len(problem))
    # print(problem)
    frontier = PQ.PriorityQueue()
    parentState.clear()
    stateCost.clear()

    currentState = initialStat
    parentState[currentState] = None
    stateCost[currentState] = 0

    print("spanning tree")
    while currentState is not None and not goalStateCheck(currentState, goal):
       for p in problem:
          newState = resultOfAction(currentState, p)
          cost = unitCost(p)

          # if type(res) != int:
          #     newState = res
          #     cost = unitCost(p)
          # else:
          #     continue
          # newState = resultOfAction(currentState, p)
          # cost = unitCost(p)
          if newState not in stateCost and newState not in frontier:
               parentState[newState] = currentState
               stateCost[newState] = stateCost[currentState] + cost
               frontier.push(newState, p = priorityFunction(newState,
currentState))
               # logging.warning(f"Added new node to frontier (cost={
stateCost[newState]})")
          elif newState in frontier and stateCost[newState] >
stateCost[currentState] + cost:
               old_cost = stateCost[currentState]
               parentState[newState] = currentState
               stateCost[newState] = stateCost[currentState] + cost
               # logging.warning(f"Updated node cost in frontier: {old_cost} ->
{stateCost[newState]}")

       if frontier:
           currentState = frontier.pop()
```

```
        else:
            currentState = None

    path = list()
    s = currentState

    while s:
        path.append(s.copyData())
        s = parentState[s]
        # path.append(s.copyData())

    logging.warning(f"Found a solution in {len(path):,} steps; visited
{len(stateCost):,} states")
    return list(reversed(path))
```

state.py

```
import
numpy


class State:
    def __init__(self, data: numpy.ndarray):
        self._data = data

    def __hash__(self):
        return hash(bytes(self._data))
    #
    def __eq__(self, other):
        return bytes(self._data) == bytes(other._data)

    def __lt__(self, other):
        return bytes(self._data) < bytes(other._data)
    #
    def __str__(self):
        return str(self._data)

    def __repr__(self):
        return repr(self._data)

    def __len__(self):
```

```
        return len(self._data)

    def contain(self):
        return self._data

    def copyData(self):
        return self._data.copy()

    def createSet (self, x, y):
        return set
```

Results achieved through the solution discovered are as below:

for N = 5:
Found a solution in 3 steps; visited 32 states
path = [{0}, {0, 1, 3}, {0, 1, 2, 3, 4}]
W = 9

for N = 10:
Found a solution in 3 steps; visited 776 states
path = [{2, 5}, {1, 2, 5, 6, 8}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}]
W = 17

for N = 20:
Found a solution in 4 steps; visited 15,887 states
path = [{0, 1, 2, 7}, {0, 1, 2, 6, 7, 9, 16, 19}, {0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 13, 14, 16, 17, 19}, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}]
W = 47

Review of Marco Masera on My Work for Lab1

Marco-Masera commented on Oct 19, 2022

Hi. I reviewed your code.

If I got it right you were trying to write an A* search algorithm. In this I found a few iusses:
-The priority function computes the intersection between the elements covered by the old state and the ones in the new state. But since creating a state means adding new elements to the existing set, this intersection always returns the set of the previous state.

This cannotfunction as an heuristic function for A* as it does not returns a lower bound for the best solution from the given Node. A possible and simple function could be computing the number of elements that still needs to be covered to get to a solution.

-UnitCost and stateCost: the unit_cost does not represent the cost of a Node, since the cost is given by the total number of elements in the lists chosen, not the number of lists itself.

Moreover, it seems like the stateCost dictionary is used in order to update the parentState dictionary. But this should not be necessary: if the A* algorithm works properly, given a certain state, you can be sure it will always be visited with the lowest cost path from the starting one, so there is no need to keep track of that.

-In the search function, when you find a node that you already visited but with a lower cost, you update the stateCost. Keep in mind that this does not update the priorityQueue.

My suggestion is to:
-Modify the States so you keep track not only of the elements covered by also the cost of this node, given by the total number of elements. Alternatively you could keep track of the lists used - which will cause more memory consumption but could allow for better performance improvements later.
-Modify the priority function so that it returns a lower bound for the best solution from the given node
-Order the priority queue with parameter priorityFunction() + real_cost_to_node (kept inside the state object as stated in point 1).
-Any node will be visited only once, and the first solution found will be (on of) the best solution(s).

Other possible performance improvements include:
-The search space you are generating right now include different permutations of the same lists set. Since order is not a matter here, you could tweak it so that for a given set of lists it only generates one node that contains this set - regardless of order.
-The State objects keeps the information about covered elements as a numpy array, but each time you do some operations you convert it to a set. You could decide to keep it as a set in the first place. Also sometimes you call copyData(), then you create a set from the copied data, and then you compute the intersection. This clones the data 3 times. You could simply call intersection() from the original set and Python will return a new set containing the result without changing the original.

My Peer Review of Marco D'Almo's Work for Lab1

It has to be mentioned that the code has recently been modified. It was not complete when I reviewed it.

milad818 commented on Oct 23, 2022

Review by Milad

Major Issues

- According to what I could understand from the code and additional comments, the code is incomplete and therefore, no conclusion is drawn of the achieved results.
- It seems like you are on the right track trying to move along the graph through the nearest vertex with the lowest cost without visiting the already visited vertices; however, graph creation and starting the problem could never take place due to some issues in implementation e.g. trying to loop through an integer or other data type that loops cannot work on.

Problem generator

Problem generator as given by professor, with default N values.

In [3]:

```
N_VALUES = [5, 10, 20, 100, 500, 1000]
SEED = 42
```

```python
import logging
import random
from copy import copy
import random
import platform
from collections import Counter
from gx_utils import *

def problem(N, seed=None):
    '''Function to generate a set covering problem with number N, default
seed is None'''
    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N //
5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]
```

Greedy solution

Greed solution as given by professor.

In [4]:

```python
import logging

def greedy(N):
    '''Just adds lists sorted from shortest to longest until the set is
covered'''
    goal = set(range(N))
    covered = set()
    solution = list()
    nodes = 0
    all_lists = sorted(problem(N, seed=SEED), key=lambda l: len(l))
    while goal != covered:
        x = all_lists.pop(0)
        nodes+=1
        if not set(x) < covered:
            solution.append(x)
            covered |= set(x)

    logging.info(
        f"Greedy solution for N={N}: w={sum(len(_) for _ in solution)}
(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%, Nodes visited:
{nodes})"
    )
    logging.debug(f"{solution}")
```

In [5]:

```python
logging.getLogger().setLevel(logging.INFO)
for N in N_VALUES:
    greedy(N)
```

```
INFO:root:Greedy solution for N=5: w=5 (bloat=0%, Nodes visited: 13)
INFO:root:Greedy solution for N=10: w=13 (bloat=30%, Nodes visited: 14)
INFO:root:Greedy solution for N=20: w=46 (bloat=130%, Nodes visited: 14)
INFO:root:Greedy solution for N=100: w=332 (bloat=232%, Nodes visited: 23)
INFO:root:Greedy solution for N=500: w=2162 (bloat=332%, Nodes visited: 28)
INFO:root:Greedy solution for N=1000: w=4652 (bloat=365%, Nodes visited: 27)
```

"longest first" Greedy

Greedy solution as seen in the professor version, just starting from the longest random list first. Performs generally worse, fairly better in just one case, N = 20. It is probably a random consequence of the list generation.

In [6]:

```
def longest_greedy(N):
    '''As the greedy, but the first list to be added is the longest'''
    goal = set(range(N))
    covered = set()
    solution = list()
    nodes=0
    all_lists = sorted(problem(N, seed=SEED), key=lambda l: len(l))
    first_iter = True
    while goal != covered:
        if first_iter:
            x = x = all_lists.pop()
            nodes+=1
            first_iter = False
        else:
            x = all_lists.pop(0)
            nodes+=1
        if not set(x) < covered:
            solution.append(x)
            covered |= set(x)

    logging.info(
            f"Alternate greedy version for N={N}: w={sum(len(_) for _ in
solution)} (bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%, Nodes
visited: {nodes})")


logging.getLogger().setLevel(logging.INFO)
for N in N_VALUES:
   longest_greedy(N)
```

```
INFO:root:Alternate greedy version for N=5: w=5 (bloat=0%, Nodes visited: 5)
INFO:root:Alternate greedy version for N=10: w=14 (bloat=40%, Nodes visited:
15)
INFO:root:Alternate greedy version for N=20: w=36 (bloat=80%, Nodes visited:
15)
INFO:root:Alternate greedy version for N=100: w=340 (bloat=240%, Nodes visite
d: 19)
INFO:root:Alternate greedy version for N=500: w=2187 (bloat=337%, Nodes visit
ed: 28)
```

```
INFO:root:Alternate greedy version for N=1000: w=4699 (bloat=370%, Nodes visi
ted: 28)
```

Different solution

Another possible solution with the implementation of Dijkstra search. Note - this is incomplete.

In [7]:

```
from gx_utils import *

class Graph:
    '''graph inizialization, compute'''
    def __init__(self, num_vert):
        self.v = num_vert
        self.edges = [[-1 for i in range(num_vert)] for j in range(num_vert)]
        self.visited = []

    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        #self.edges[v][u] = weight

def create_graph(graph, problem_list):
    for i in range(graph.v):
        for j in range(graph.v):
            graph.add_edge(i, j, len(problem_list[j]))
            graph.add_edge(j, i, len(problem_list[i]))
```

In [8]:

```
def dijkstra_sol(N):

    GOAL = set(range(N))

    def problem_start(N):
        'starts the problem, adds the first vertex, so the shortest list, to
the visited'
        p = sorted(problem(N, seed=SEED), key=lambda l: len(l))
        g = Graph(len(p))
        create_graph(g, p)

        return g, p

    def visited_to_set(state, problem_list):
        sol = [tuple(problem_list[e]) for e in state]
        return set(sum((e for e in sol), start=()))

    def goal_test(state, problem_list):
        return visited_to_set(state, problem_list) == GOAL

    _, p = problem_start(N)

    D = {v:float('inf') for v in range(_.v)}
    solution = p
```

```
    for start in range(_.v):
        g, p = problem_start(N)
        pq = PriorityQueue()
        pq.push((0, start), p=0)
        #pq.put((0, start))
        D = {v:float('inf') for v in range(_.v)}
        D[start]=0

        while pq:
            (dist, current_vertex) = pq.pop()
            g.visited.append(current_vertex)

            for neighbor in range(g.v):

                if g.edges[current_vertex][neighbor] != -1:
                    new_elements = visited_to_set(g.visited, p) -
set(p[neighbor])
                    distance = g.edges[current_vertex][neighbor] -
len(new_elements)


                    if neighbor not in g.visited:
                        old_cost = D[neighbor]
                        cost = D[current_vertex] + distance

                        if cost < old_cost:
                            pq.push((cost, neighbor), p=cost)
                            D[neighbor] = cost

                    if goal_test(g.visited, p):

                        new_solution = [p[elem] for elem in g.visited]
                        weight_new = sum(len(_) for _ in new_solution)
                        weight_old = sum(len(_) for _ in solution)
                        solution = new_solution if weight_new<weight_old else
solution

                        if weight_new==N:
                            return solution

                        pq = PriorityQueue()

    return solution

# incomplete and possibly non functioning approach, left it here to leave
trace of what was delivered
# will probably complete/review it for learning purposes
```

In [15]:

```
logging.getLogger().setLevel(logging.DEBUG)

for N in [5, 10, 20]:
    solution = dijkstra_sol(N)
```

```
    logging.info(
        f" Solution for N={N:,}: "
        + f"w={sum(len(_) for _ in solution):,} "
        + f"(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%), solution
is {solution}"
    )
```

```
INFO:root: Solution for N=5: w=5 (bloat=0%), solution is [[0], [1], [4], [2],
[3]]
INFO:root: Solution for N=10: w=10 (bloat=0%), solution is [[8, 9, 3], [6], [
0, 4], [2, 5], [1, 7]]
INFO:root: Solution for N=20: w=33 (bloat=65%), solution is [[6, 9, 11, 12, 1
7], [8, 4, 7], [18, 2, 15], [8, 16, 5], [1, 3, 13, 14], [17, 18, 7], [17, 10,
1, 7], [16, 9, 19, 6], [0, 1, 2, 7]]
```

NOTE: everything here is copied from professor Squillero's solution and is here for the sake of completion of the exercise and to keep track of the different algorithms. It should not be considered as mine.

Dijkstra's

Professor Squillero's solution, with added comments for learning purposes.

>
>
> https://github.com/squillero/computational-intelligence
>
>

In [10]:

```
from gx_utils import *

def dijkstra(N, all_lists):
    """Vanilla Dijkstra's algorithm"""

    # Goal and utility variables initialization
    # all_lists is the problem generated sorted list
    GOAL = set(range(N))

    # generates a tuple consisting of the lists of the problem
    # ordered and unchangeable list
    all_lists = tuple(set(tuple(_) for _ in all_lists))
    frontier = PriorityQueue()
    nodes = 0

    # converts a state (list of visited nodes) into a set
    # to be able to compare with goal
    def state_to_set(state):
        return set(sum((e for e in state), start=()))

    # comparison with goal, returns bool
    def goal_test(state):
```

```
        return state_to_set(state) == GOAL

    # returns a list of lists, which
    # could be the possible step -> all except
    # subsets or equivalent sets
    def possible_steps(state):
        current = state_to_set(state)
        return [l for l in all_lists if not set(l) <= current]

    # computation of the internal weight
    # per state
    def w(state):
        cnt = Counter()
        cnt.update(sum((e for e in state), start=()))
        return sum(cnt[c] - 1 for c in cnt if cnt[c] > 1), -sum(cnt[c] == 1
for c in cnt)

    # keeps on updating the frontier based on
    # internal weight computed by w() function
    # and then choosing the next state as the one with
    # the lowest internal weight
    state = tuple()
    while state is not None and not goal_test(state):
        nodes += 1
        for s in possible_steps(state):
            frontier.push((*state, s), p=w((*state, s)))
        state = frontier.pop()

    logging.debug(f"dijkstra: SOLVED! nodes={nodes:,}; w={sum(len(_) for _ in
state):,}; iw={w(state)})")
    return state
```

In [14]:

```
logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20]:
    solution = dijkstra(N, problem(N, seed=42))
    logging.info(
        f" Solution for N={N:,}: "
        + f"w={sum(len(_) for _ in solution):,} "
        + f"(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
    )
```

```
INFO:root: Solution for N=5: w=5 (bloat=0%)
INFO:root: Solution for N=10: w=10 (bloat=0%)
INFO:root: Solution for N=20: w=23 (bloat=15%)
```

Hill Climbing

Professor Squillero Hill Climbing solution, with added comments for learning purposes.

https://github.com/squillero/computational-intelligence

In [20]:

```
def hc(N, all_lists):
    """Vanilla Hill Climber"""

    # same operation as for Dijkstra
    all_lists = set(tuple(_) for _ in all_lists)

    # state evaluation by number of lists in the counter
    # and total sum of the elements in them
    def evaluate(state):
        cnt = Counter()
        cnt.update(sum((e for e in state), start=()))
        return len(cnt), -cnt.total()

    # tweak function on the current solution
    # randomly eliminates a list from the solution
    # or adds a list non present in the solution
    # from the possible lists
    def tweak(solution):
        new_solution = set(solution)
        while new_solution and random.random() < 0.7:
            r = random.choice(list(new_solution))
            new_solution.remove(r)
        while random.random() < 0.7:
            a = random.choice(list(all_lists - solution))
            new_solution.add(a)
        return new_solution

    # sets a max number of consecutive useless steps
    # to stop tweaking
    current_solution = set()
    useless_steps = 0
    while useless_steps < 10_000:
        useless_steps += 1
        candidate_solution = tweak(current_solution)
        if evaluate(candidate_solution) > evaluate(current_solution):
            useless_steps = 0
            current_solution = copy(candidate_solution)
            logging.debug(f"New solution: {evaluate(current_solution)}")
    return current_solution
```

In [30]:

```
logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20, 100, 1000]:
    solution = hc(N, problem(N, seed=42))
    logging.info(
        f" Solution for N={N:,}: "
        + f"w={sum(len(_) for _ in solution):,} "
```

```
        + f"(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
    )
```

```
INFO:root: Solution for N=5: w=5 (bloat=0%)
INFO:root: Solution for N=10: w=11 (bloat=10%)
INFO:root: Solution for N=20: w=24 (bloat=20%)
INFO:root: Solution for N=100: w=214 (bloat=114%)
INFO:root: Solution for N=1,000: w=3,383 (bloat=238%)
```

In [16]:

```
def hc_adapt(N, all_lists):
    """Vanilla Hill Climber"""

    # same operation as for Dijkstra
    all_lists = set(tuple(_) for _ in all_lists)

    # state evaluation by number of lists in the counter
    # and total sum of the elements in them
    def evaluate(state):
        cnt = Counter()
        cnt.update(sum((e for e in state), start=()))
        return len(cnt), -cnt.total()

    # tweak function on the current solution
    # randomly eliminates a list from the solution
    # or adds a list non present in the solution
    # from the possible lists
    def tweak0(solution):
        new_solution = set(solution)
        while new_solution and random.random() < 0.7:
            r = random.choice(list(new_solution))
            new_solution.remove(r)
        while random.random() < 0.7:
            a = random.choice(list(all_lists - solution))
            new_solution.add(a)
        return new_solution

    def tweak1(solution):
        new_solution = set(solution)
        while new_solution and random.random() < 0.8:
            r = random.choice(sorted(list(new_solution), key=lambda x:
len(x))[0:5])
            new_solution.remove(r)
        while random.random() < 0.8:
            a = random.choice(list(all_lists - solution))
            new_solution.add(a)
        return new_solution

    def tweak2(solution):
        new_solution = set(solution)
        while new_solution and random.random() < 0.8:
            r = sorted(list(new_solution), key=lambda x: len(x))[0]
            new_solution.remove(r)
            a = random.choice(list(all_lists - solution))
```

```python
            new_solution.add(a)
        return new_solution

    # sets a max number of consecutive useless steps
    # to stop tweaking
    current_solution = set()
    useless_steps = 0
    while useless_steps < 10_000:
        useless_steps += 1

        if useless_steps < 5000:
            candidate_solution = tweak0(current_solution)
        elif 5000 < useless_steps < 8000:
            candidate_solution = tweak1(current_solution)
        else:
            candidate_solution = tweak2(current_solution)

        if evaluate(candidate_solution) > evaluate(current_solution):
            useless_steps = 0
            current_solution = copy(candidate_solution)
            logging.debug(f"New solution: {evaluate(current_solution)}")
    return current_solution
```

In [29]:

```python
logging.getLogger().setLevel(logging.INFO)

for N in [5, 10, 20, 100, 500, 1000]:
    solution = hc_adapt(N, problem(N, seed=42))
    logging.info(
        f" Solution for N={N:,}: "
        + f"w={sum(len(_) for _ in solution):,} "
        + f"(bloat={(sum(len(_) for _ in solution)-N)/N*100:.0f}%)"
    )
```

```
INFO:root: Solution for N=5: w=5 (bloat=0%)
INFO:root: Solution for N=10: w=10 (bloat=0%)
INFO:root: Solution for N=20: w=24 (bloat=20%)
INFO:root: Solution for N=100: w=214 (bloat=114%)
INFO:root: Solution for N=500: w=1,509 (bloat=202%)
INFO:root: Solution for N=1,000: w=3,469 (bloat=247%)
```

Lab 2

Task

Given a number N and some lists of integers $P = (L_0, L_1, L_2, \ldots, L_n)$, determine, if possible, $S = (L_{s_0}, L_{s_1}, L_{s_2}, \ldots, L_{s_n})$ such that each number between 0 and $N-1$ appears in at least one list

$$\forall n \in [0, N-1]\ \exists i : n \in L_{s_i}$$

and that the total numbers of elements in all L_{s_i} is minimum.

Group Members: Milad Zakhireh (300708), Masoud Karimi (300283)

---

main.py

```
import
problem
          import myEA
          seed = 42
          Ns = [5, 10, 20, 100, 500, 1000]
          populationSizes = [10, 10, 10, 20, 30, 200]
          problemSizes = [4, 4, 4, 7, 10, 13]
          offspringSizes = [5, 5, 5, 10, 15, 50]
          generatorsSize = 500
          for index in range(len(Ns)):
              myProblem = problem.Problem(Ns[index], seed)
              ea = myEA.EA(problemSizes[index], populationSizes[index],
          offspringSizes[index], generatorsSize, myProblem.setOfProblem)
              print(ea.resultFitness)
              print(ea.coveredLists)
```

myEA.py

```
import
copy
         from collections import namedtuple, Counter
         import random
         from functools import cmp_to_key
         import numpy

         Individual = namedtuple("Individual", ["genome", "fitness"])
         class EA:
```

```
    def __init__(self, problemSize, populationSize, offspringSize, generatorsSize,
setOfProblem):
       self.problemsize = problemSize
       self.populationSize = populationSize
       self.offspringSize = offspringSize
       self.generatorsSize = generatorsSize
       self.listOfIndevidualsAndTheirFitness = list()
       self.rankedListOfIndevidualsAndTheirFitness = list()
       self.roulletWheelOfIndevidualsAndTheirFitness = list()
       self.setOfProblem = setOfProblem
       self.creatingPopulation(self.setOfProblem)
       self.breeding()
       self.resultIndex = self.rankedListOfIndevidualsAndTheirFitness[0].genome
       self.resultFitness = self.rankedListOfIndevidualsAndTheirFitness[0].fitness
       self.coveredLists = numpy.array(self.setOfProblem,
dtype=set)[list(self.resultIndex)]




    def creatingPopulation(self, setOfProblem ):
        # Individual = namedtuple("Individual", ["genome", "fitness"])
        for genome in [tuple([ random.choice(range(len(setOfProblem))) for _ in
range(self.problemsize)]) for __ in range(self.populationSize) ]:
            self.listOfIndevidualsAndTheirFitness.append(Individual(genome,
self.forEvaluation(genome, setOfProblem)))
        self.rankOfEachIndevidualByMeansOfRouletteWheelApproach()
        self.rankedAsRoulletWheel()

    def compare(self,pair1, pair2):
        _, fitness1 = pair1
        _, fitness2 = pair2
        digits1, total1 = fitness1
        digits2, total2 = fitness2

        if digits1 == digits2:
            if total1 < total2:
                return -1
            else:
                return 1
        if digits1 < digits2:
            return -1
```

```
        else:
            return 1

    def rankOfEachIndevidualByMeansOfRouletteWheelApproach(self):
        self.rankedListOfIndevidualsAndTheirFitness =
sorted(self.listOfIndevidualsAndTheirFitness, key=cmp_to_key(self.compare),reverse
= True)

    def rankedAsRoulletWheel(self):
        length = len(self.rankedListOfIndevidualsAndTheirFitness)
        for eachIndevidual in
range(len(self.rankedListOfIndevidualsAndTheirFitness)):
           genome, _ = self.rankedListOfIndevidualsAndTheirFitness[eachIndevidual]
           length-=1

self.roulletWheelOfIndevidualsAndTheirFitness.append(Individual(genome,length))



    def forEvaluation(self, genome, setOfProblem=None):
        localList = list()
        if setOfProblem is not None:
            for index in genome:
                localList.append(setOfProblem[index])
        else:
            for index in genome:
                localList.append(self.setOfProblem[index])
        cnt = Counter()
        cnt.update(sum((e for e in localList), start=()))
        return len(cnt), -cnt.total()

    def breeding(self):
        for g in range(self.generatorsSize):
            offspring = list()
            for i in range(self.offspringSize):
                if random.random() < 0.3:
                    p = self.tournament()
                    o = self.mutation(p.genome)
                    # print(o)
                else:
                    p1 = self.tournament()
                    p2 = self.tournament()
                    o = self.cross_over(p1.genome, p2.genome)
```

```
                    # print(o)
                f = self.forEvaluation(o)
                offspring.append(Individual(o, f))
            self.listOfIndevidualsAndTheirFitness += offspring
            self.rankOfEachIndevidualByMeansOfRouletteWheelApproach()
            self.rankedAsRoulletWheel()

            self.rankedListOfIndevidualsAndTheirFitness =
self.rankedListOfIndevidualsAndTheirFitness[:self.populationSize]
            self.roulletWheelOfIndevidualsAndTheirFitness =
self.roulletWheelOfIndevidualsAndTheirFitness[:self.populationSize]

    def tournament(self, tournament_size=2):
        return max(random.choices(self.roulletWheelOfIndevidualsAndTheirFitness,
k=tournament_size), key=lambda i: i.fitness)

    def cross_over(self, g1, g2):
        cut = random.randint(0, self.problemsize)
        return g1[:cut] + g2[cut:]

    def mutation(self, g):
        g= list(g)
        point = random.randint(0, self.problemsize - 1)
        indexForMutation = random.randint(0,len(self.setOfProblem)-1)
        return tuple(g[:point] + [indexForMutation] + g[point + 1:])
```

problem.py

```
import
random

class Problem:

    def __init__(self, n, seed):
        self.prombelSize = n
        self.listOfProblem = self.myProblem(n, seed)
        self.setOfProblem = self.creatSetFromListOfProblem()

    def myProblem(self, N, seed=None):
        """Creates an instance of the problem"""
```

```
        random.seed(seed)
        return [
            list(set(random.randint(0, N - 1) for n in range(random.randint(N // 5, N // 2))))
            for n in range(random.randint(N, N * 5))
        ]

    def creatSetFromListOfProblem(self):
        sortedList = set()
        for _ in self.listOfProblem:
            sortedelement  = sorted(_)
            sortedList.add(tuple(sortedelement))
        return sorted(sortedList)
```

Results

N	weight	nodes
5	5	[(3,) (1,) (4,) (0, 2)]
10	10	[(2, 5) (1, 7) (0, 4) (3, 6, 8, 9)]
20	29	[(2, 6, 8, 10, 12, 15, 18) (0, 1, 3, 7, 9, 10, 11, 15)(1, 3, 8, 11, 14, 19) (4, 5, 8, 13, 15, 16, 17, 19)]
100	244	many nodes
500	1739	many nodes
1000	3980	many nodes

Review of Marco Masera on My Work for Lab 2

Marco-Masera commented on Nov 13, 2022 • edited

Hi. I leave the review for Lab2

General considerations

It's a good implementation of GA for this problem. The code is very clean and readable, I like this stile of writing.

Major iusses

I did not find any major iusses in the code or in the solution.

Suggestions

These are not iusses but suggestions. You can try to play around a little bit more with different phases during the execution of the algorithm.
I did too write my algorithm with no phases - meaning it follows the same rules for every generation - and I found that after some iterations the individuals become too much similar one with another, and the solution stops improving. After the presentation of last monday I realised how much the result can improve with a finer control of the balance between exploration and exploitation.

Conclusion

I don't have much to say except for: good job :)

My Peer Review of Shayan Taghinezhad Roudbaraki's Work for Lab2

milad818 commented on Nov 12, 2022

Review by Milad

In general, the implementation is good and there seem to be some innovative ideas behind it. However, it is worth mentioning some issues which should have been given more attention.

Major Issues

- I could understand from your code that you have only employed the crossover operator and no mutation has been carried out which could possibly lead to more efficient results.
- It was kind of challenging to understand how the variable N has been used and I failed to discover the logic behind the corresponding lines.
- Results are not included in the readme file

Minor Issues

- Despite the nice implementation of the idea in mind, the code is not so much readable. Using a reasonable number of comments could be helpful.

```
import random, logging
seed = 42
N = 10

logging.getLogger().setLevel(logging.DEBUG)

def problem(N, seed=None):
    """Creates an instance of the problem"""

    random.seed(seed)
    return [
        list(set(random.randint(0, N - 1) for n in range(random.randint(N // 
5, N // 2))))
        for n in range(random.randint(N, N * 5))
    ]


all_lists = problem(10, 42)



logging.debug(f"all_lists is {all_lists}")
```

```
DEBUG:root:all_lists is [[0, 4], [1, 2, 3], [9, 6], [0, 1], [8, 9, 3], [8, 3]
, [0, 3, 4, 7, 9], [4, 5, 6], [1, 3, 5], [1, 6], [0, 9, 4, 5], [8, 1, 6], [9,
3, 5], [0, 3], [1, 3, 6], [2, 5, 7], [1, 3, 4, 9], [8, 2, 3], [3, 4, 5, 6, 8]
, [0, 3], [1, 3, 4, 6], [3, 6, 7], [2, 3, 4], [9, 6], [8, 2, 3, 7], [0, 1], [
9, 2, 6], [6], [8, 0, 4, 1], [1, 4, 5, 6], [0, 4, 7], [8, 1, 4], [2, 5], [9,
5], [0, 1, 3, 4, 5], [9, 3], [1, 7], [8, 2], [8, 2, 7], [8, 9, 3, 6], [4, 5,
```

```
6], [8, 1, 3, 7], [0, 5], [0, 9, 3], [0, 3], [0, 5], [8, 3], [8, 2, 3, 7], [1
, 3, 6, 7], [5, 6]]
```

In [51]:

```
def fitness(genome, n=N):
    correct_n = len(set(genome))
    repeated_n = len(genome) - correct_n
    return correct_n - repeated_n

def cross_over(g1, g2):
    return g1 + g2

def tournament(population, tournament_size=2):
    return max(random.choices(population, k=tournament_size), key=lambda i:
i.fitness)
```

In [52]:

```
from collections import namedtuple
from collections import Counter

Individual = namedtuple("Individual", ["genome", "fitness"])
population = list()

for i in range(10):
    genome = random.choice(all_lists)
    indv = Individual(genome, fitness(genome, 10))
    population.append(indv)

logging.debug(f"poulation is {population}")
```

```
DEBUG:root:poulation is [Individual(genome=[6], fitness=1), Individual(genome
=[9, 2, 6], fitness=3), Individual(genome=[1, 4, 5, 6], fitness=4), Individua
l(genome=[8, 3], fitness=2), Individual(genome=[0, 1], fitness=2), Individual
(genome=[0, 9, 3], fitness=3), Individual(genome=[8, 1, 3, 7], fitness=4), In
dividual(genome=[8, 1, 3, 7], fitness=4), Individual(genome=[0, 3, 4, 7, 9],
fitness=5), Individual(genome=[0, 1], fitness=2)]
```

In [54]:

```
logging.getLogger().setLevel(logging.INFO)

NUM_GENERATIONS = 100
OFFSPRING_SIZE = 3
POPULATION_SIZE = 10

for g in range(NUM_GENERATIONS):
    offspring = list()
    for i in range(OFFSPRING_SIZE):
        p1 = tournament(population)
        p2 = tournament(population)
        o = cross_over(p1.genome, p2.genome)
        f = fitness(o)
        offspring.append(Individual(o, f))
    population += offspring
    population = sorted(population, key=lambda i: i.fitness,
reverse=True)[:POPULATION_SIZE]
```

```
logging.info(f"population is {population}")
```

```
INFO:root:population is [Individual(genome=[1, 4, 5, 6, 0, 3, 4, 7, 9], fitne
ss=7), Individual(genome=[1, 4, 5, 6, 0, 3, 4, 7, 9], fitness=7), Individual(
genome=[8, 3, 0, 3, 4, 7, 9, 9, 2, 6], fitness=6), Individual(genome=[1, 4, 5
, 6, 8, 3, 0, 3, 4, 7, 9, 9, 2, 6], fitness=6), Individual(genome=[0, 3, 4, 7
, 9, 9, 2, 6, 1, 4, 5, 6], fitness=6), Individual(genome=[8, 3, 0, 3, 4, 7, 9
, 9, 2, 6, 1, 4, 5, 6], fitness=6), Individual(genome=[9, 2, 6, 1, 4, 5, 6, 0
, 3, 4, 7, 9], fitness=6), Individual(genome=[0, 3, 4, 7, 9], fitness=5), Ind
ividual(genome=[0, 1, 0, 3, 4, 7, 9], fitness=5), Individual(genome=[9, 2, 6,
1, 4, 5, 6], fitness=5)]
```

Lab 3

Task

Write agents able to play *Nim*, with an arbitrary number of rows and an upper bound k on the number of objects that can be removed in a turn (a.k.a., *subtraction game*).

The player **taking the last object wins**.

- Task3.1: An agent using fixed rules based on *nim-sum* (i.e., an *expert system*)
- Task3.2: An agent using evolved rules
- Task3.3: An agent using minmax
- Task3.4: An agent using reinforcement learning

Instructions

- Create the directory `lab3` inside the course repo
- Put a `README.md` and your solution (all the files, code and auxiliary data if needed)

Group Members: Milad Zakhireh (300708), Masoud Karimi (300283)

main.py

```
from
Nim
import
MyNim
        from RowObjectsPair import RowObjectsPair
        from copy import deepcopy
        x = MyNim(3, 1)
        s0 = (0,1)
        r0, o0 = s0
        srtgy0 = RowObjectsPair(r0, o0)
        # 3.1 & 3.2
        # MyNim.doNimming(x,{"p":0.61})
```

```
# ---------------------------------------------------
print(x.inferedStatus["possible_moves"])
# 3.3
state0 =deepcopy( x._rows )
# x.inferedStatus["possible_moves"]-= {s0}
state1 = x.staticNimming(x._rows, srtgy0)
print(x.minMax(state0, state1))
```

Nim.py

```
import
itertools
from operator import xor
import random
from RowObjectsPair import RowObjectsPair
from copy import deepcopy
from typing import Callable


class MyNim:

    def __init__(self, numRows : int, k : int = None )-> None:
        self.countOfRows = numRows
        self._rows = [(i*2)+1 for i in range(numRows)]
        self.copyOfRows = list()
        self._k = k
        self.inferedStatus = dict()
        self.inferedStatusForCopiedRows = dict()
        self.inferableInformation()

    def __bool__(self):
        return sum(self._rows) > 0

    def __str__(self):
        return "<" + " ".join(str(_) for _ in self._rows) + " >"

    @property
    def rows(self) -> tuple:
        return tuple(self._rows)
```

```
    @property
    def k(self) -> int:
        return self._k

    def deepCopyOf_rows(self):
        self.copyOfRows = deepcopy(self._rows)
        return self.copyOfRows

# 3.1, 3,2
    def nimming(self, strategy :RowObjectsPair, ifFindingNimSum = None) ->
None:
        if not ifFindingNimSum:
            row =  strategy.row
            numObjects = strategy.numberOfObject
            # assert self._rows[row] >= numObjects
            # assert self._k is None or numObjects <= self._k
            self._rows[row] -= numObjects
        else:
            row = strategy.row
            numObjects = strategy.numberOfObject
            self.copyOfRows[row] -= numObjects

    def staticNimming(self, rows, strategy :RowObjectsPair):
        row = strategy.row
        numObjects = strategy.numberOfObject
        rows[row] -= numObjects
        return deepcopy(rows)

    def doNimming(self, gemone:dict = None):
        if gemone is not None:
            strategies = [self.properActionToPerformNimSum,
self.evaluate(gemone)]
        else:
            strategies = [self.properActionToPerformNimSum,
self.pureRandomStrategy]
        print(f"initial board : {self._rows}")
        player = 0
        while self:
            # if not player:
            #     strategy = strategies[player]()
            # else:
            strategy = strategies[player]()
            self.nimming(strategy)
```

```
            print(f" PLayer: {player}, Board: {self._rows}")
            player = 1 - player
        print(f"winner is player {player}")


    def inferableInformation(self, evaluation = False) -> None:
        if not evaluation:
            self.inferedStatus["oddRows"] = [count  for count,rowValue in
enumerate(self._rows) if rowValue%2]
            self.inferedStatus["evenRows"] = [count for count, rowValue in
enumerate(self._rows) if not rowValue % 2]
            self.inferedStatus["nimSum"] = self.nimSum()
            self.inferedStatus["longets"] = self._rows.index(max(self._rows,
key= lambda i:i))
            # self.inferedStatus["shortest"] = self._rows.index(min(i for i in
self._rows if i > 0))
            self.inferedStatus["ifRowsAreEven"] = len([count  for
count,rowValue in enumerate(self._rows) if rowValue%2])%2 == 0
            self.inferedStatus["possible_moves"] = set([(r, a1) for r,i in
enumerate(self._rows) for a1 in range(i + 1) if a1 >= self.k and a1 <= a1 + 1])
        else:
            self.inferedStatusForCopiedRows["oddRows"] = [count for count,
rowValue in enumerate(self._rows) if rowValue % 2]
            self.inferedStatusForCopiedRows["evenRows"] = [count for count,
rowValue in enumerate(self._rows) if not rowValue % 2]
            self.inferedStatusForCopiedRows["nimSum"] = self.nimSum()
            self.inferedStatusForCopiedRows["longets"] =
self._rows.index(max(self._rows, key=lambda i: i))
            # self.inferedStatusForCopiedRows["shortest"] =
self._rows.index(min(i for i in self._rows if i > 0))
            self.inferedStatusForCopiedRows["ifRowsAreEven"] = len([count for
count, rowValue in enumerate(self._rows) if rowValue % 2]) % 2 == 0
            self.inferedStatusForCopiedRows["possible_moves"] = set([(r, a1)
for r, i in enumerate(self._rows) for a1 in range(i + 1) if a1 >= self.k and a1
<= a1 + 1])


    def pureRandomStrategy(self)-> RowObjectsPair:
        selectedRow = random.choice([row for row, count in
enumerate(self._rows) if count > 0 ])
        numberOfObjects = random.randint(1,self._rows[selectedRow])
        return RowObjectsPair(selectedRow, numberOfObjects)
```

```
    def pureRandomStrategyPluseWhole(self)-> RowObjectsPair:
        selectedRow = random.choice([row for row, count in
enumerate(self._rows) if count > 0 ])
        numberOfObjects = self._rows[selectedRow]
        return RowObjectsPair(selectedRow, numberOfObjects)


    def nimSum(self, ifFindingNimSum = None) -> list:
        if not ifFindingNimSum:
            *_, result = itertools.accumulate(self._rows, xor)
            return result
        else:
            *_, result = itertools.accumulate(self.copyOfRows, xor)
            return result


    def properActionToPerformNimSum(self) -> RowObjectsPair:
       self.inferableInformation()
       x = self.inferedStatus["nimSum"]
       if x:
            possibleMoves = self.inferedStatus["possible_moves"]

            for rowActionPair in possibleMoves:
                _ = self.deepCopyOf_rows()
                indexOfPossibleObjects, possibleObjects = rowActionPair
                self.nimming(RowObjectsPair(indexOfPossibleObjects,
possibleObjects), self.copyOfRows)
                if  not self.nimSum(self.copyOfRows):
                    self.copyOfRows = list()
                    return RowObjectsPair(indexOfPossibleObjects,
possibleObjects)

            self.copyOfRows = list()
            return self.pureRandomStrategy()
       self.copyOfRows = list()
       return self.pureRandomStrategy()

    def evaluate(self, selectionThreshold: dict) -> Callable:
        def evolvable() -> RowObjectsPair:
            self.inferableInformation()
            if random.random() < selectionThreshold["p"]:
                ply = RowObjectsPair(self.inferedStatus["shortest"],
random.randint(1, self._rows[self.inferedStatus["shortest"]]))
```

```
            else:
                ply = RowObjectsPair(self.inferedStatus["longets"],
random.randint(1, self._rows[self.inferedStatus["longets"]]))
            return ply
        return evolvable


#3.3 MIN MAX

    def ifWinner(self, playerState):
        if playerState.count(0) == (self.countOfRows-1):
            return True



    def evaluateMinMAx(self, p0,p1):

        if self.ifWinner(p0):
            return 1
        elif self.ifWinner(p1):
            return -1
        else:
            return 0



    def minMax(self, s0, s1):
        self.inferableInformation()
        possibleActions = self.inferedStatus["possible_moves"]
        eval = self.evaluateMinMAx(s0, s1)
        if eval != 0:
            return eval

        evaluation = list()
        for possibleAction in possibleActions :
            s0 = deepcopy(s1)
            if s0[possibleAction[0]]>= possibleAction[1]:
                s0 = self.staticNimming(s0, RowObjectsPair(possibleAction[0],
possibleAction[1]))
            else:
                continue
            eval = self.minMax(s1, s0)
            if type(eval) == tuple:
                _, eval = eval
            evaluation.append((possibleAction,-eval))
```

```
                return max(evaluation, key=lambda k: k[1])
```

RowObjectsPair.py

```
class
RowObjectsPair:
                    def __init__(self, row, numberOfObject):
                        self.row = row
                        self.numberOfObject = numberOfObject

                    def __str__(self):
                        return "<STRATEGY= row: " + str(self.row) + ", objects: " +
                str(self.numberOfObject)+ ">"
```

Review of Marco Masera on My Work for Lab 3

Marco-Masera commented on Dec 18, 2022

Hi. I leave the review for lab3.

General considerations

The lack of documentation and comments in the code makes it hard to understand how the code works; the choice of having a single class implement almost all of the logic, while being an interesting design choice, makes it even harder.

It would also be useful to have some tests to run.

Perhaps you were planning to add these later, in case I'll update the review.

That said I liked the idea of having a more "generic" agent and having evolving rules and nimsum as simple strategies nested inside it. It's probably not the cleanest or more readable way to do it, but it's elegant in a way.

Nim Sum:

It's a working implementation of the NimSum strategy for games with K = None, while it does not extend to games with K defined. You could implement NimSum for games with K defined by computing the xor-sum on a transformed list where each value if the module of the number of elements in the row and k+1: [n % (k+1) for n in rows]

Evolving rules:

The implementation of the evolving rules part seems incomplete. The class implements it with a genome dict that contains a single value **p**, specifying the probability of choosing between two different rules. But the code that manages the creation and evolution of individuals isn't there.

Min Max:

I find this implementation confusing and I'm not 100% sure it's correct.
I don't really understand why it takes two states as input, and when recursively calling itself it pass its second argument as first argument to the next call, while the new status is passed as the second argument. The most straightforward implementation would be to pass one state of the game to the function. Again I think it would be really useful to provide some comments or documentation to explain these choices.

Reinforcement learning agent:

Missing.

Conclusion:

This review isn't the most useful; unfortunately I had a hard time navigating inside the code without documentation, comments or tests to run.

The project of course is not complete so this makes the review even less useful.

What I can say is that NimSum seems correct, the evolving rules part seems *on the good rail* but needs to have the part of the code that actually does the evolution; you can start from the code you wrote for Lab2, which was good.
You could also experiment with more rules to choose from or a more complex combination of different rules.

The MinMax part is a little bit confusing and I'd suggest writing a more straightforward minmax recursive function.

Final Project (Quarto)

The project is implemented with the guidance of my dear friend Masoud Karimi who helped me so much to improve my background knowledge alongside with the lectures all throughout the course.

main.py

```
# Free for
personal or
classroom
use; see
'LICENSE.md'
for details.
# https://github.com/squillero/computational-intelligence

import logging
import argparse
```

```
import random
import quarto
import numpy
from numpy import save
import matplotlib.pyplot as plot
import copy


class RandomPlayer(quarto.Player):
    """Random player"""

    def __init__(self, quarto: quarto.Quarto, learningPhase) -> None:
        super().__init__(quarto,learningPhase)

    def choose_piece(self) -> int:
        return random.randint(0, 15)

    def place_piece(self) -> tuple[int, int]:
        return random.randint(0, 3), random.randint(0, 3)

def main():
    # game = quarto.Quarto()
    # playerRL = quarto.Player(game, True)
    # playerR = RandomPlayer(game, True)
    # game.set_players((playerRL, playerR))
    # game.getAvailablePieces()
    # game.getFreePlaces()
    # placeReward, pieceReward =
game.learnModelParams(copy.deepcopy(game.availablePieces))
    # save("finalPlaceWeight2.npy", placeReward)
    # save("finalPieceWeight2.npy", pieceReward)

    pieceReward = numpy.load('finalPieceWeight2.npy', allow_pickle=True)
    pieceReward = dict(enumerate(pieceReward.flatten(), 1))
    pieceReward = pieceReward[1]
    placeReward = numpy.load('finalPlaceWeight2.npy', allow_pickle=True)
    placeReward = dict(enumerate(placeReward.flatten(), 1))
    placeReward = placeReward[1]
    print(pieceReward)
    print(placeReward)

    # ----------------------------------------------------/\ Battle Field
/\----------------------------------------------
```

```
    rounds = 800
    runIndex = 10
    RlProportion = []
    Randomproportion = []
    valueX = []

    for j in range(runIndex):
        RL = 0
        rand = 0
        draw = 0
        for i in range(rounds):
            battleField = quarto.Quarto()
            agentRL = quarto.Player(battleField, False)
            agentRandom = RandomPlayer(battleField, False)
            agentRL.pieceWeightDict = pieceReward
            agentRL.placeWeightDict = placeReward
            battleField.set_players((agentRL, agentRandom))
            winner = battleField.run()
            if winner == 0:
                RL += 1
            elif winner == 1:
                rand += 1
            else:
                draw += 1
        valueX.append(j)
        Proportion = RL / rounds
        RlProportion.append(Proportion)
        Randomproportion.append(1 - Proportion)
        print(f"RL rate ={RL / rounds} and randon rate = {rand / rounds}")

    plot.semilogy(valueX, RlProportion, "b")
    plot.axhline(y=0.5, color='r', linestyle='--')
    plot.xlim([-1.0, runIndex])
    plot.ylim([0, 1])
    plot.title("Precentage Of RL Agent Wins")
    plot.legend(["RL agent", "Mean"])
    plot.xlabel("Runs")
    plot.ylabel(f"{rounds}-Rounds")

    plot.show()


if __name__ == '__main__':
```

```
    parser = argparse.ArgumentParser()
    parser.add_argument('-v', '--verbose', action='count', default=0,
help='increase log verbosity')
    parser.add_argument('-d',
                        '--debug',
                        action='store_const',
                        dest='verbose',
                        const=2,
                        help='log debug messages (same as -vv)')
    args = parser.parse_args()

    if args.verbose == 0:
        logging.getLogger().setLevel(level=logging.WARNING)
    elif args.verbose == 1:
        logging.getLogger().setLevel(level=logging.INFO)
    elif args.verbose == 2:
        logging.getLogger().setLevel(level=logging.DEBUG)

    main()
```

quarto/objects.py

```
# Free for
personal or
classroom
use; see
'LICENSE.md'
for details.
# https://github.com/squillero/computational-intelligence
import numpy
import numpy as np
from abc import abstractmethod
import copy
import random
from functools import cmp_to_key


class Player(object):
```

```
    def __init__(self, quarto, learningPhase, rounds=500, alpha=0.2,
randomFactor=0.1) -> None:
        self.__quarto = quarto
        # self.__board = quarto.get_board_status
        self.historyOfMoves = []  # place, reward
        self.historyOfPiece = []  # piece, reward
        self.alpha = alpha
        self.randomFactor = randomFactor
        self.placeWeightDict = {}
        self.pieceWeightDict = {}
        self.learningPhase = learningPhase
        self.setReward(self.__quarto)
        self.rounds = rounds
        self.currentRound = 0
        self.tempSelected = set()

    def setReward(self, quarto):
        if self.learningPhase:
            for i, row in enumerate(quarto.get_board_status()):
                for j, col in enumerate(row):
                    self.placeWeightDict[(i, j)] =
np.random.uniform(low=1.0, high=0.1)

            for pI,_ in enumerate(quarto.get_all_pieces()):
                self.pieceWeightDict[pI] = np.random.uniform(low=1.0,
high=0.1)


    @abstractmethod
    def choose_piece(self) -> int:
        # self.__quarto.boa
        self.__quarto.getAvailablePieces()
        maxG = -10e15
        maxGrule = -10e9

        next_piece_index = None
        randomN = np.random.random()
        if self.learningPhase:

            if randomN < self.randomFactor:
                next_piece_index = random.randint(0, 15)

            else:
```

```
                pieceWeight = []
                for pieceIndex in self.__quarto.availablePieces:
                    selectedPieceWeight =
self.__quarto.calcPieceWeight(pieceIndex)
                    pieceWeight.append((pieceIndex, selectedPieceWeight))

                sortedList = sorted(pieceWeight,
key=cmp_to_key(self.compare), reverse=True)
                # the worst
                next_piece_index_help, _ = sortedList[0]
                # next_piece_index = next_piece_index_help

                for pieceIndex, weight in pieceWeight:
                    new_piece_index = pieceIndex
                    tempW = weight
                    if self.pieceWeightDict[new_piece_index] >= maxG or
tempW >= maxGrule:
                        next_piece_index = new_piece_index
                        maxG = self.pieceWeightDict[pieceIndex]
                        maxGrule = tempW

                if next_piece_index is None:
                    print("help piece")
                    next_piece_index = next_piece_index_help




        else:
            for pieceIndex in self.__quarto.availablePieces:
                new_piece_index = pieceIndex
                if self.pieceWeightDict[new_piece_index] >= maxG:
                    next_piece_index = new_piece_index
                    maxG = self.pieceWeightDict[pieceIndex]

        return next_piece_index

    def update_player_board(self):
        self.__board = self.__quarto.get_board_status

    def compare(self, pair1, pair2):
        _, fitness1 = pair1
        _, fitness2 = pair2
```

```
        if fitness2 > fitness1:
            return -1
        else:
            return 1

    @abstractmethod
    def place_piece(self) -> tuple[int, int]:
        # self.__board= self.__quarto.get_board_status()
        pI = self.__quarto.get_selected_piece()
        self.__quarto.updatePlayedPiecePlace()
        self.__quarto.getFreePlaces()
        maxG = -10e15
        maxGrun = -10e15
        next_move = None
        randomN = np.random.random()

        if self.learningPhase:

            if randomN < self.randomFactor:
                freePLaces = [tuple(i) for i in self.__quarto.allFreePlaces]
                next_move = random.choice(freePLaces)
            else:
                pairXYfeasibility = []
                for placeXY in self.__quarto.allFreePlaces:
                    weight = self.__quarto.calcPlaceWeight(pI, placeXY)
                    pairXYfeasibility.append((placeXY, weight))
                sortedList = sorted(pairXYfeasibility,
key=cmp_to_key(self.compare), reverse=True)
                next_move_help, _ = sortedList[0]
                next_move = next_move_help

                for action, palceWeight in pairXYfeasibility:
                    new_state = action
                    # and palceWeight >= maxGrun
                    if self.placeWeightDict[new_state] >= maxG and
palceWeight >= maxGrun:
                        next_move = new_state
                        maxG = self.placeWeightDict[new_state]
                        maxGrun = palceWeight

                if next_move is None:
                    print("place help")
                    next_move = next_move_help
```

```python
        else:
            for action in self.__quarto.allFreePlaces:
                new_state = action
                if self.placeWeightDict[new_state] >= maxG:
                    next_move = new_state
                    maxG = self.placeWeightDict[new_state]
        return next_move

    def get_game(self):
        return self.__quarto

    # def updateMovesHistory(self, place):
    #     if self.__quarto.assignReward() != 0:
    #         reward = 1 / self.__quarto.assignReward()
    #     else:
    #         reward =  self.__quarto.assignReward()
    #     self.historyOfMoves.append((place, reward))

    def updateHistoryOfMoves(self, place):
        reward = self.__quarto.assignReward()
        self.historyOfMoves.append((place, reward))


    def updateHistoryOfPiece(self, piece):
        reward = self.__quarto.assignReward()
        self.historyOfPiece.append((piece, reward))


class CharacCounter(object):
    def __init__(self, selectedPieceCharacteristic = None):
        if selectedPieceCharacteristic == None:
            self.high = 0
            self.notHigh = 0
            self.colored = 0
            self.notColored = 0
            self.solid = 0
            self.notSolid = 0
            self.square = 0
            self.notSquare = 0
        else:
            if selectedPieceCharacteristic.HIGH:
                self.high = 1
```

```
                self.notHigh = 0
            else:
                self.notHigh = 1
                self.high = 0

            if selectedPieceCharacteristic.COLOURED:
                self.colored = 1
                self.notColored = 0
            else:
                self.notColored = 1
                self.colored = 0

            if selectedPieceCharacteristic.SOLID:
                self.solid = 1
                self.notSolid = 0
            else:
                self.notSolid = 1
                self.solid = 0

            if selectedPieceCharacteristic.SQUARE:
                self.square = 1
                self.notSquare = 0
            else:
                self.notSquare = 1
                self.square = 0

class Piece(object):

    def __init__(self, high: bool, coloured: bool, solid: bool, square:
bool) -> None:
        self.HIGH = high
        self.COLOURED = coloured
        self.SOLID = solid
        self.SQUARE = square


class Quarto(object):

    MAX_PLAYERS = 2
    BOARD_SIDE = 4

    def __init__(self) -> None:
        self.__players = ()
```

```
        self.reset()


    def reset(self):
        self.__board = np.ones(shape=(self.BOARD_SIDE, self.BOARD_SIDE),
dtype=int) * -1
        self.__pieces = []
        self.__pieces.append(Piece(False, False, False, False))  # 0
        self.__pieces.append(Piece(False, False, False, True))  # 1
        self.__pieces.append(Piece(False, False, True, False))  # 2
        self.__pieces.append(Piece(False, False, True, True))  # 3
        self.__pieces.append(Piece(False, True, False, False))  # 4
        self.__pieces.append(Piece(False, True, False, True))  # 5
        self.__pieces.append(Piece(False, True, True, False))  # 6
        self.__pieces.append(Piece(False, True, True, True))  # 7
        self.__pieces.append(Piece(True, False, False, False))  # 8
        self.__pieces.append(Piece(True, False, False, True))  # 9
        self.__pieces.append(Piece(True, False, True, False))  # 10
        self.__pieces.append(Piece(True, False, True, True))  # 11
        self.__pieces.append(Piece(True, True, False, False))  # 12
        self.__pieces.append(Piece(True, True, False, True))  # 13
        self.__pieces.append(Piece(True, True, True, False))  # 14
        self.__pieces.append(Piece(True, True, True, True))  # 15
        self.__current_player = 0
        self.__selected_piece_index = -1
        self.getFreePlaces()
        self.learningPhase = False
        self.allFreePlaces = None
        self.availablePieces = None

    def set_players(self, players: tuple[Player, Player]):
        self.__players = players

    def updatePlayedPiecePlace(self):
        self.getAvailablePieces()
        self.getFreePlaces()

    def getAvailablePieces(self):
        allIndices = set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
15])
        board = self.get_board_status().ravel()
        allSelectedPieces = set(board[board > -1])
        self.availablePieces = list(allIndices - allSelectedPieces)
```

```
    def getFreePlaces(self):
        self.freePlaces = np.where(self.__board == -1)
        self.allFreePlaces = zip(self.freePlaces[0], self.freePlaces[1])


    def compare(self, pair1, pair2):
        _, fitness1 = pair1
        _, fitness2 = pair2

        if fitness2 > fitness1:
            return -1
        else:
            return 1

    def select(self, pieceIndex: int) -> bool:
        '''
        select a piece. Returns True on success
        '''
        if pieceIndex not in self.__board:
            self.__selected_piece_index = pieceIndex
            return True
        return False

    def place(self, x: int, y: int) -> bool:
        '''
        Place piece in coordinates (x, y). Returns true on success
        '''
        if self.__placeable(x, y):
            self.__board[x, y] = self.__selected_piece_index
            return True
        return False

    def __placeable(self, x: int, y: int) -> bool:
        return not (y < 0 or x < 0 or x > 3 or y > 3 or self.__board[x, y]
>= 0)

    def print(self):
        '''
        Print the __board
        '''
        for row in self.__board:
            print("\n -------------------")
```

```
                print("|", end="")
                for element in row:
                    print(f" {element: >2}", end=" |")
            print("\n -------------------\n")
            print(f"Selected piece: {self.__selected_piece_index}\n")

    def get_piece_charachteristics(self, index: int) -> Piece:
        '''
        Gets charachteristics of a piece (index-based)
        '''
        return copy.deepcopy(self.__pieces[index])

    def get_board_status(self) -> np.ndarray:
        '''
        Get the current __board status (__pieces are represented by index)
        '''
        return copy.deepcopy(self.__board)

    def get_all_pieces(self):
        return copy.deepcopy(self.__pieces)

    def get_selected_piece(self) -> int:
        '''
        Get index of selected piece
        '''
        return copy.deepcopy(self.__selected_piece_index)

    def __check_horizontal(self) -> int:
        hDict = dict()
        hDict[0] = None
        hDict[1] = None
        hDict[2] = None
        hDict[3] = None

        for i in range(self.BOARD_SIDE):
            initList = CharacCounter()

            high_values = [
                elem for elem in self.__board[i] if elem >= 0 and
self.__pieces[elem].HIGH
            ]
            initList.high = len(high_values)
```

```
            coloured_values = [
                elem for elem in self.__board[i] if elem >= 0 and
self.__pieces[elem].COLOURED
            ]
            initList.colored = len(coloured_values)

            solid_values = [
                elem for elem in self.__board[i] if elem >= 0 and
self.__pieces[elem].SOLID
            ]
            initList.solid = len(solid_values)

            square_values = [
                elem for elem in self.__board[i] if elem >= 0 and
self.__pieces[elem].SQUARE
            ]
            initList.square = len(square_values)

            low_values = [
                elem for elem in self.__board[i] if elem >= 0 and not
self.__pieces[elem].HIGH
            ]
            initList.notHigh = len(low_values)

            noncolor_values = [
                elem for elem in self.__board[i] if elem >= 0 and not
self.__pieces[elem].COLOURED
            ]
            initList.notColored = len(noncolor_values)

            hollow_values = [
                elem for elem in self.__board[i] if elem >= 0 and not
self.__pieces[elem].SOLID
            ]
            initList.notSolid = len(hollow_values)

            circle_values = [
                elem for elem in self.__board[i] if elem >= 0 and not
self.__pieces[elem].SQUARE
            ]
            initList.notSquare = len(circle_values)
            hDict[i] = initList
```

```
            if len(high_values) == self.BOARD_SIDE or len(
                    coloured_values
            ) == self.BOARD_SIDE or len(solid_values) == self.BOARD_SIDE or
len(
                square_values) == self.BOARD_SIDE or len(low_values) ==
self.BOARD_SIDE or len(
                noncolor_values) == self.BOARD_SIDE or len(
                hollow_values) == self.BOARD_SIDE or len(
                circle_values) == self.BOARD_SIDE:
                return self.__current_player, None

        return -1, hDict

    def __check_vertical(self):
        vDict = dict()
        vDict[0] = None
        vDict[1] = None
        vDict[2] = None
        vDict[3] = None

        for i in range(self.BOARD_SIDE):
            # counts the total value of hight are selected
            initList = CharacCounter()
            high_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and
self.__pieces[elem].HIGH
            ]
            initList.high = len(high_values)

            coloured_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and
self.__pieces[elem].COLOURED
            ]
            initList.colored = len(coloured_values)

            solid_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and
self.__pieces[elem].SOLID
            ]
            initList.solid = len(solid_values)

            square_values = [
```

```
                elem for elem in self.__board[:, i] if elem >= 0 and
self.__pieces[elem].SQUARE
            ]
            initList.square = len(square_values)

            low_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and not
self.__pieces[elem].HIGH
            ]
            initList.notHigh = len(low_values)

            noncolor_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and not
self.__pieces[elem].COLOURED
            ]
            initList.notColored = len(noncolor_values)

            hollow_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and not
self.__pieces[elem].SOLID
            ]
            initList.notSolid = len(hollow_values)

            circle_values = [
                elem for elem in self.__board[:, i] if elem >= 0 and not
self.__pieces[elem].SQUARE
            ]
            initList.notSquare = len(circle_values)

            vDict[i] = initList

            if len(high_values) == self.BOARD_SIDE or len(
                    coloured_values
            ) == self.BOARD_SIDE or len(solid_values) == self.BOARD_SIDE or
len(
                square_values) == self.BOARD_SIDE or len(low_values) ==
self.BOARD_SIDE or len(
                noncolor_values) == self.BOARD_SIDE or len(
                hollow_values) == self.BOARD_SIDE or len(
                circle_values) == self.BOARD_SIDE:
                return self.__current_player, None
        return -1, vDict
```

```
def new_check_diagonal(self):
    LToRdiagDict = dict()
    LToRdiagDict[0] = None
    LToRdiagDict[1] = None
    LToRdiagDict[2] = None
    LToRdiagDict[3] = None

    for i in range(self.BOARD_SIDE):
        # if self.__board[i, i] < 0:
        #     break
        high_values = []
        coloured_values = []
        solid_values = []
        square_values = []
        low_values = []
        noncolor_values = []
        hollow_values = []
        circle_values = []
        LTRinitiallist = CharacCounter()

        if self.__pieces[self.__board[i, i]].HIGH:
            if self.__board[i, i] != -1:
                high_values.append(self.__board[i, i])
                LTRinitiallist.high = len(high_values)
        else:
            if self.__board[i, i] != -1:
                low_values.append(self.__board[i, i])
                LTRinitiallist.notHigh = len(low_values)

        if self.__pieces[self.__board[i, i]].COLOURED:
            if self.__board[i, i] != -1:
                coloured_values.append(self.__board[i, i])
                LTRinitiallist.colored = len(coloured_values)
        else:
            if self.__board[i, i] != -1:
                noncolor_values.append(self.__board[i, i])
                LTRinitiallist.notColored = len(noncolor_values)

        if self.__pieces[self.__board[i, i]].SOLID:
            if self.__board[i, i] != -1:
                solid_values.append(self.__board[i, i])
                LTRinitiallist.solid = len(solid_values)
        else:
```

```
                if self.__board[i, i] != -1:
                    hollow_values.append(self.__board[i, i])
                    LTRinitiallist.notSolid = len(hollow_values)

            if self.__pieces[self.__board[i, i]].SQUARE:
                if self.__board[i, i] != -1:
                    square_values.append(self.__board[i, i])
                    LTRinitiallist.square = len(square_values)
            else:
                if self.__board[i, i] != -1:
                    circle_values.append(self.__board[i, i])
                    LTRinitiallist.notSquare = len(circle_values)

            LToRdiagDict[i] = LTRinitiallist

        if len(high_values) == self.BOARD_SIDE or len(coloured_values) ==
self.BOARD_SIDE or len(
                solid_values) == self.BOARD_SIDE or len(square_values) ==
self.BOARD_SIDE or len(
            low_values
        ) == self.BOARD_SIDE or len(noncolor_values) == self.BOARD_SIDE or
len(
            hollow_values) == self.BOARD_SIDE or len(circle_values) ==
self.BOARD_SIDE:
            return self.__current_player, None

        RToLdiagDict = dict()
        RToLdiagDict[0] = None
        RToLdiagDict[1] = None
        RToLdiagDict[2] = None
        RToLdiagDict[3] = None

        for i in range(self.BOARD_SIDE):
            # if self.__board[i, self.BOARD_SIDE - 1 - i] < 0:
            #     break
            high_values = []
            coloured_values = []
            solid_values = []
            square_values = []
            low_values = []
            noncolor_values = []
            hollow_values = []
            circle_values = []
```

```
            RTLinitiallist = CharacCounter()
            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 - i]].HIGH:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    high_values.append(self.__board[i, self.BOARD_SIDE - 1 -
i])
                    RTLinitiallist.high = len(high_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    low_values.append(self.__board[i, self.BOARD_SIDE - 1 -
i])
                    RTLinitiallist.notHigh = len(low_values)

            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 -
i]].COLOURED:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    coloured_values.append(
                        self.__board[i, self.BOARD_SIDE - 1 - i])
                    RTLinitiallist.colored = len(coloured_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    noncolor_values.append(
                        self.__board[i, self.BOARD_SIDE - 1 - i])
                    RTLinitiallist.notColored = len(noncolor_values)

            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 -
i]].SOLID:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    solid_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.solid = len(solid_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    hollow_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.notSolid = len(hollow_values)

            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 -
i]].SQUARE:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    square_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.square = len(square_values)
            else:
```

```
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    circle_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.notSquare = len(circle_values)

            RToLdiagDict[i] = RTLinitiallist

        if len(high_values) == self.BOARD_SIDE or len(coloured_values) ==
self.BOARD_SIDE or len(
                solid_values) == self.BOARD_SIDE or len(square_values) ==
self.BOARD_SIDE or len(
            low_values
        ) == self.BOARD_SIDE or len(noncolor_values) == self.BOARD_SIDE or
len(
            hollow_values) == self.BOARD_SIDE or len(circle_values) ==
self.BOARD_SIDE:
            return self.__current_player, None

        retunDict = {
            "LTR": LToRdiagDict,
            "RTL": RToLdiagDict
        }
        return -1, retunDict

    def calcPieceWeight(self, pieceIndex):
        badPrise = -10e10
        copyOfBoard = copy.deepcopy(self.__board)
        self.updatePlayedPiecePlace()
        thisFreePlacesRewardForGivenPiece = []
        for possition in self.allFreePlaces:
            x, y = possition
            self.__board = copy.deepcopy(copyOfBoard)
            self.__board[x, y] = pieceIndex
            _, dictH = self.__check_horizontal()
            _, dictV = self.__check_vertical()
            _, dictD = self.new_check_diagonal()
            dictLTR, dictRTL = dictD["LTR"], dictD["RTL"]
            if dictH is not None and dictV is not None and dictLTR is not
None and dictRTL is not None:
                propertyH = dictH[x]
                propertyV = dictV[y]
                thisStepH = numpy.array(
```

```
                    [propertyH.high, propertyH.notHigh, propertyH.colored,
propertyH.notColored, propertyH.solid,
                     propertyH.notSolid, propertyH.square,
                     propertyH.notSquare])
                thisStepV = numpy.array(
                    [propertyV.high, propertyV.notHigh, propertyV.colored,
propertyV.notColored, propertyV.solid,
                     propertyV.notSolid, propertyV.square,
                     propertyV.notSquare])
                l1 = dictLTR[0]
                l2 = dictLTR[1]
                l3 = dictLTR[2]
                l4 = dictLTR[3]
                thisStepL = numpy.array(
                    [l1.high + l2.high + l3.high + l4.high, l1.notHigh +
l2.notHigh + l3.notHigh + l4.notHigh,
                     l1.colored + l2.colored + l3.colored + l4.colored,
                     l1.notColored + l2.notColored + l3.notColored +
l4.notColored,
                     l1.solid + l2.solid + l3.solid + l4.solid, l1.notSolid
+ l2.notSolid + l3.notSolid + l4.notSolid,
                     l1.square + l2.square + l3.square + l4.square,
                     l1.notSquare + l2.notSquare + l3.notSquare +
l4.notSquare])

                r1 = dictRTL[0]
                r2 = dictRTL[1]
                r3 = dictRTL[2]
                r4 = dictRTL[3]
                thisStepR = numpy.array(
                    [r1.high + r2.high + r3.high + r4.high, r1.notHigh +
r2.notHigh + r3.notHigh + r4.notHigh,
                     r1.colored + r2.colored + r3.colored + r4.colored,
                     r1.notColored + r2.notColored + r3.notColored +
r4.notColored,
                     r1.solid + r2.solid + r3.solid + r4.solid, r1.notSolid
+ r2.notSolid + r3.notSolid + r4.notSolid,
                     r1.square + r2.square + r3.square + r4.square,
                     r1.notSquare + r2.notSquare + r3.notSquare +
r4.notSquare])

                h4 = len(thisStepH[thisStepH == 4])
                h3 = len(thisStepH[thisStepH == 3])
```

```
                h2 = len(thisStepH[thisStepH == 2])
                h1 = len(thisStepH[thisStepH == 1])
                h0 = len(thisStepH[thisStepH == 0])

                v4 = len(thisStepV[thisStepV == 4])
                v3 = len(thisStepV[thisStepV == 3])
                v2 = len(thisStepV[thisStepV == 2])
                v1 = len(thisStepV[thisStepV == 1])
                v0 = len(thisStepV[thisStepV == 0])

                l4 = len(thisStepL[thisStepL == 4])
                l3 = len(thisStepL[thisStepL == 3])
                l2 = len(thisStepL[thisStepL == 2])
                l1 = len(thisStepL[thisStepL == 1])
                l0 = len(thisStepL[thisStepL == 0])

                r4 = len(thisStepR[thisStepR == 4])
                r3 = len(thisStepR[thisStepR == 3])
                r2 = len(thisStepR[thisStepR == 2])
                r1 = len(thisStepR[thisStepR == 1])
                r0 = len(thisStepR[thisStepR == 0])

                temp = [h4, h3, h2, h1, h0, v4, v3, v2, v1, v0, l4, l3, l2,
l1, l0, r4, r3, r2, r1, r0]
                forNormalize = numpy.array(temp)
                forNormalize = forNormalize[forNormalize != 0]
                totalValued = sum(forNormalize)

                if r4 != 0 or l4 != 0 or v4 != 0 or h4 != 0:
                    thisFreePlacesRewardForGivenPiece.append(((x, y),
badPrise))
                else:
                    newPrise = 5 * (h3 + v3 + l3 + r3) + 2 * (h2 + v2 + l2 +
r2) + 1 * (h1 + v1 + l1 + r1) + 1 * (
                            h0 + v0 + l0 + r0)

                    newPrise = newPrise / totalValued
                    thisFreePlacesRewardForGivenPiece.append(((x, y),
newPrise))
            else:
                self.__board = copy.deepcopy(copyOfBoard)
                return badPrise
```

```
        sortedList = sorted(thisFreePlacesRewardForGivenPiece,
key=cmp_to_key(self.compare), reverse=True)
        worstPlaceForGivenPiece, worstPriseForGivenPiece = sortedList[0]
        self.__board = copy.deepcopy(copyOfBoard)
        return worstPriseForGivenPiece

    def __check_diagonal(self):
        LTRdiagDict = dict()
        LTRdiagDict[0] = None
        LTRdiagDict[1] = None
        LTRdiagDict[2] = None
        LTRdiagDict[3] = None
        high_values = []
        coloured_values = []
        solid_values = []
        square_values = []
        low_values = []
        noncolor_values = []
        hollow_values = []
        circle_values = []

        for i in range(self.BOARD_SIDE):
            # if self.__board[i, i] < 0:
            #     break
            LTRinitiallist = CharacCounter()

            if self.__pieces[self.__board[i, i]].HIGH:
                if self.__board[i, i] != -1:
                    high_values.append(self.__board[i, i])
                    LTRinitiallist.high = len(high_values)
            else:
                if self.__board[i, i] != -1:
                    low_values.append(self.__board[i, i])
                    LTRinitiallist.notHigh = len(low_values)

            if self.__pieces[self.__board[i, i]].COLOURED:
                if self.__board[i, i] != -1:
                    coloured_values.append(self.__board[i, i])
                    LTRinitiallist.colored = len(coloured_values)
            else:
                if self.__board[i, i] != -1:
                    noncolor_values.append(self.__board[i, i])
                    LTRinitiallist.notColored = len(noncolor_values)
```

```
            if self.__pieces[self.__board[i, i]].SOLID:
                if self.__board[i, i] != -1:
                    solid_values.append(self.__board[i, i])
                    LTRinitiallist.solid = len(solid_values)
            else:
                if self.__board[i, i] != -1:
                    hollow_values.append(self.__board[i, i])
                    LTRinitiallist.notSolid = len(hollow_values)

            if self.__pieces[self.__board[i, i]].SQUARE:
                if self.__board[i, i] != -1:
                    square_values.append(self.__board[i, i])
                    LTRinitiallist.square = len(square_values)
            else:
                if self.__board[i, i] != -1:
                    circle_values.append(self.__board[i, i])
                    LTRinitiallist.notSquare = len(circle_values)

            LTRdiagDict[i] = LTRinitiallist

        if len(high_values) == self.BOARD_SIDE or len(coloured_values) ==
self.BOARD_SIDE or len(
                solid_values) == self.BOARD_SIDE or len(square_values) ==
self.BOARD_SIDE or len(
            low_values
        ) == self.BOARD_SIDE or len(noncolor_values) == self.BOARD_SIDE or
len(
            hollow_values) == self.BOARD_SIDE or len(circle_values) ==
self.BOARD_SIDE:
            return self.__current_player, None

        RTLdiagDict = dict()
        RTLdiagDict[0] = None
        RTLdiagDict[1] = None
        RTLdiagDict[2] = None
        RTLdiagDict[3] = None
        high_values = []
        coloured_values = []
        solid_values = []
        square_values = []
        low_values = []
        noncolor_values = []
```

```
        hollow_values = []
        circle_values = []

        for i in range(self.BOARD_SIDE):
            # if self.__board[i, self.BOARD_SIDE - 1 - i] < 0:
            #     break
            RTLinitiallist = CharacCounter()
            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 - i]].HIGH:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    high_values.append(self.__board[i, self.BOARD_SIDE - 1 -
i])
                    RTLinitiallist.high = len(high_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    low_values.append(self.__board[i, self.BOARD_SIDE - 1 -
i])
                    RTLinitiallist.notHigh = len(low_values)

            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 -
i]].COLOURED:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    coloured_values.append(
                        self.__board[i, self.BOARD_SIDE - 1 - i])
                    RTLinitiallist.colored = len(coloured_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    noncolor_values.append(
                        self.__board[i, self.BOARD_SIDE - 1 - i])
                    RTLinitiallist.notColored = len(noncolor_values)

            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 -
i]].SOLID:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    solid_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.solid = len(solid_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    hollow_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.notSolid = len(hollow_values)
```

```
            if self.__pieces[self.__board[i, self.BOARD_SIDE - 1 -
i]].SQUARE:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    square_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.square = len(square_values)
            else:
                if self.__board[i, self.BOARD_SIDE - 1 - i] != -1:
                    circle_values.append(self.__board[i, self.BOARD_SIDE - 1
- i])
                    RTLinitiallist.notSquare = len(circle_values)

            RTLdiagDict[i] = RTLinitiallist

        if len(high_values) == self.BOARD_SIDE or len(coloured_values) ==
self.BOARD_SIDE or len(
                solid_values) == self.BOARD_SIDE or len(square_values) ==
self.BOARD_SIDE or len(
            low_values
        ) == self.BOARD_SIDE or len(noncolor_values) == self.BOARD_SIDE or
len(
            hollow_values) == self.BOARD_SIDE or len(circle_values) ==
self.BOARD_SIDE:
            return self.__current_player, None

        retunDict = {
            "LTR": LTRdiagDict,
            "RTL": RTLdiagDict
        }
        return -1, retunDict



    def assignReward(self):
        if not self.check_finished():
            return -100 * int(not self.check_finished())
        else:
            return 10e2

    def check_winner(self) -> int:
        '''
        Check who is the winner
        '''
```

```
        checkV, _ = self.__check_vertical()
        checkH, _ = self.__check_horizontal()
        checkD, _ = self.__check_diagonal()

        l = [checkH, checkV, checkD]
        # l = [self.__check_horizontal(), self.__check_vertical(),
self.__check_diagonal()]
        for elem in l:
            if elem >= 0:
                return elem
        return -1

    def check_finished(self) -> bool:
        '''
        Check who is the loser
        '''
        for row in self.__board:
            for elem in row:
                if elem == -1:
                    return False
        return True

    def learnModelParams(self, pieces):
        agentRL = 0
        agentRandom = 0
        draw = 0
        if self.__players[0].learningPhase:
            print("inlearing ")

            for epoch in range(self.__players[0].rounds):
                self.__board = np.ones(shape=(self.BOARD_SIDE,
self.BOARD_SIDE), dtype=int) * -1
                self.availablePieces = pieces
                winner = -1

                ++self.__players[0].currentRound
                while winner < 0 and not self.check_finished():

                    self.updatePlayedPiecePlace()
                    piece_ok = False
                    while not piece_ok:
                        self.updatePlayedPiecePlace()
```

```
                        selectedPiece =
self.__players[self.__current_player].choose_piece()
                        piece_ok = self.select(selectedPiece)
                        if self.__players[0].learningPhase:
                            if piece_ok and not bool(self.__current_player):

self.__players[0].updateHistoryOfPiece(self.__selected_piece_index)
                    piece_ok = False
                    self.__current_player = (self.__current_player + 1) %
self.MAX_PLAYERS
                    while not piece_ok:
                        self.updatePlayedPiecePlace()
                        place =
self.__players[self.__current_player].place_piece()
                        x, y = place
                        piece_ok = self.place(x, y)
                        if self.__players[0].learningPhase:
                            if piece_ok and not bool(self.__current_player):

self.__players[0].updateHistoryOfMoves(place)
                    winner = self.check_winner()
                print(f"the winner is ={winner}")
                if winner == 0:
                    agentRL += 1
                elif winner == 1:
                    agentRandom += 1
                else:
                    draw += 1
                # or winner == -1
                if winner == 0 :
                    self.learn(self.__players[0])
                else:
                    self.__players[0].historyOfMoves = []
                    self.__players[0].historyOfPiece = []



            print(f"RL wins: {agentRL} and Random wins: {agentRandom} and
Draw is: {draw}, ")
            return self.__players[0].placeWeightDict,
self.__players[0].pieceWeightDict

    def learn(self, player):
        target = 0
```

```
        for prev, reward in reversed(player.historyOfMoves):
            player.placeWeightDict[prev] = player.placeWeightDict[prev] +
player.alpha * (target - player.placeWeightDict[prev])
            target += reward

        target = 0
        for prev, reward in reversed(player.historyOfPiece):
            player.pieceWeightDict[prev] = player.pieceWeightDict[prev] +
player.alpha * (target - player.pieceWeightDict[prev])
            target += reward

        player.historyOfMoves= []
        player.historyOfPiece = []

        player.randomFactor -= 10e-5  # decrease random factor each episode
of play

    def calcPlaceWeight(self, pieceIndex, placeXY):
        grandPrise = 10e10
        worstPrise = -grandPrise
        # 2 file 5
        thresholdFor3 = 2
        copyOfBoard = self.get_board_status()
        self.updatePlayedPiecePlace()
        thisFreePlacesRewardForGivenPiece = []

        x, y = placeXY
        self.__board = copy.deepcopy(copyOfBoard)
        self.__board[x, y] = pieceIndex
        _, dictH = self.__check_horizontal()
        _, dictV = self.__check_vertical()
        _, dictD = self.new_check_diagonal()
        dictLTR, dictRTL = dictD["LTR"], dictD["RTL"]
        if dictH is not None and dictV is not None and dictLTR is not None
and dictRTL is not None:
            propertyH = dictH[x]
            propertyV = dictV[y]
            thisStepH = numpy.array(
                [propertyH.high, propertyH.notHigh, propertyH.colored,
propertyH.notColored, propertyH.solid,
                 propertyH.notSolid, propertyH.square,
                 propertyH.notSquare])
            thisStepV = numpy.array(
```

```
                [propertyV.high, propertyV.notHigh, propertyV.colored,
propertyV.notColored, propertyV.solid,
                 propertyV.notSolid, propertyV.square,
                 propertyV.notSquare])
            l1 = dictLTR[0]
            l2 = dictLTR[1]
            l3 = dictLTR[2]
            l4 = dictLTR[3]
            thisStepL = numpy.array(
                [l1.high + l2.high + l3.high + l4.high, l1.notHigh +
l2.notHigh + l3.notHigh + l4.notHigh,
                 l1.colored + l2.colored + l3.colored + l4.colored,
                 l1.notColored + l2.notColored + l3.notColored +
l4.notColored,
                 l1.solid + l2.solid + l3.solid + l4.solid, l1.notSolid +
l2.notSolid + l3.notSolid + l4.notSolid,
                 l1.square + l2.square + l3.square + l4.square,
                 l1.notSquare + l2.notSquare + l3.notSquare + l4.notSquare])

            r1 = dictRTL[0]
            r2 = dictRTL[1]
            r3 = dictRTL[2]
            r4 = dictRTL[3]
            thisStepR = numpy.array(
                [r1.high + r2.high + r3.high + r4.high, r1.notHigh +
r2.notHigh + r3.notHigh + r4.notHigh,
                 r1.colored + r2.colored + r3.colored + r4.colored,
                 r1.notColored + r2.notColored + r3.notColored +
r4.notColored,
                 r1.solid + r2.solid + r3.solid + r4.solid, r1.notSolid +
r2.notSolid + r3.notSolid + r4.notSolid,
                 r1.square + r2.square + r3.square + r4.square,
                 r1.notSquare + r2.notSquare + r3.notSquare + r4.notSquare])

            h4 = len(thisStepH[thisStepH == 4])
            h3 = len(thisStepH[thisStepH == 3])
            h2 = len(thisStepH[thisStepH == 2])
            h1 = len(thisStepH[thisStepH == 1])
            h0 = len(thisStepH[thisStepH == 0])

            v4 = len(thisStepV[thisStepV == 4])
            v3 = len(thisStepV[thisStepV == 3])
            v2 = len(thisStepV[thisStepV == 2])
```

```
            v1 = len(thisStepV[thisStepV == 1])
            v0 = len(thisStepV[thisStepV == 0])

            l4 = len(thisStepL[thisStepL == 4])
            l3 = len(thisStepL[thisStepL == 3])
            l2 = len(thisStepL[thisStepL == 2])
            l1 = len(thisStepL[thisStepL == 1])
            l0 = len(thisStepL[thisStepL == 0])

            r4 = len(thisStepR[thisStepR == 4])
            r3 = len(thisStepR[thisStepR == 3])
            r2 = len(thisStepR[thisStepR == 2])
            r1 = len(thisStepR[thisStepR == 1])
            r0 = len(thisStepR[thisStepR == 0])

            temp = [h4, h3, h2, h1, h0, v4, v3, v2, v1, v0, l4, l3, l2, l1,
l0, r4, r3, r2, r1, r0]
            forNormalize = numpy.array(temp)
            forNormalize = forNormalize[forNormalize != 0]
            totalValued = sum(forNormalize)

            if r4 != 0 or l4 != 0 or v4 != 0 or h4 != 0:
                thisFreePlacesRewardForGivenPiece.append(((x, y),
grandPrise))
            else:
                if h3 + v3 + l3 + r3 > thresholdFor3:
                    thisFreePlacesRewardForGivenPiece.append(((x, y),
worstPrise))
                else:
                    newPrise = -1 * (h3 + v3 + l3 + r3) + 2 * (h2 + v2 + l2
+ r2) + 3 * (h1 + v1 + l1 + r1) + 5 * (
                            h0 + v0 + l0 + r0)
                    newPrise = newPrise / totalValued
                    thisFreePlacesRewardForGivenPiece.append(((x, y),
newPrise))

        else:
            self.__board = copyOfBoard
            return grandPrise

        # sortedList = sorted(thisFreePlacesRewardForGivenPiece,
key=cmp_to_key(self.compare), reverse=True)
```

```
        bestPlaceForGivenPiece, bestPriseForGivenPiece =
thisFreePlacesRewardForGivenPiece[0]
        self.__board = copyOfBoard
        return bestPriseForGivenPiece

    def run(self) -> int:
        '''
        Run the game (with output for every move)
        '''
        winner = -1
        while winner < 0 and not self.check_finished():
            # self.print()
            piece_ok = False
            while not piece_ok:
                piece_ok =
self.select(self.__players[self.__current_player].choose_piece())
            piece_ok = False
            self.__current_player = (self.__current_player + 1) %
self.MAX_PLAYERS
            # self.print()
            while not piece_ok:
                x, y = self.__players[self.__current_player].place_piece()
                piece_ok = self.place(x, y)
            winner = self.check_winner()
        # self.print()
        return winner
```