

Masoud Karimi, Student ID: 300283

Laboratories and Final project report

Github repository for laboratories : <https://github.com/msdkarimi/CI-300283>

Lab #1

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
```

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))

```

Property.py

```

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

```

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)) ) ]

```

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

```

Received review for my implementation of lab#1

Wrong representation #3



Open LeonardoTredese opened this issue on Oct 20, 2022 · 0 comments



LeonardoTredese commented on Oct 20, 2022




Major

- You are [representing your state](#) as the set of different numbers you have encountered so far. This cannot work with the given [problem](#). In this approach you cannot keep track of the lists that are part of [your solution](#). This probably leads your program to add the same list twice and does not allow you to keep track of the actual solution.
- The unit cost of an action would be the number of elements in the list you are considering, it is [not unitary](#). (Ex. adding [1, 3, 7] or [2, 5] increases your solution weight respectively of 3 and 2, not always 1)
- The [priority function](#) should recall either [Dijkstra](#) (Prioritize the least costly state) or [A*](#) (Prioritize the least costly and heuristically closer state to the solution) to guarantee that you find the solution of minimum cost. Here it looks like you are performing a greedy search, which has no guarantee of finding the optimal solution.

Minor

- Please use the `__init__.py` file only if you intend to export a module.
- Avoid pushing your `__pycache__` and `.idea` directories by creating a `.gitignore` file. You can find [precompiled ones already provided by github](#).

Class used only once #2

 Open LeonardoTredese opened this issue on Oct 20, 2022 · 0 comments




LeonardoTredese commented on Oct 20, 2022 · edited ▾



Minor

The [Property class](#) is used [only once](#) to store names that are already accessible.

Useless class #1

 Open LeonardoTredese opened this issue on Oct 20, 2022 · 0 comments



LeonardoTredese commented on Oct 20, 2022 · edited ▾




Minor

Based on your use of the [State class](#), it can be substituted with a [frozenset](#), which is the fixed and hashable version of set. I suggest to avoid the definition of the hash method on a mutable object.

My reviews for lab#1

lab1 review #4

 Open msdkarimi opened this issue on Oct 24, 2022 · 0 comments



msdkarimi commented on Oct 24, 2022



1. By inspecting your code I understood that you tired to use a random list before using your greedy approach.
2. In my opinion it would be better if you try to give a cost to each list in list of lists based on its length to give priority to each list and try to pick a list based on the Lowes cost in priority queue.

kind regards,
MSD

lab1 review #2



msdkarimi opened this issue on Oct 24, 2022 · 0 comments



msdkarimi commented on Oct 24, 2022



- Well representation of the solution and also implementation, the results shows that the solution that you proposed is good enough to solve the problem for $N=50$, whilst for our solution for $N=20$ gets stuck, its another reason that shows your implementation is good enough.

regards,
MSD

Lab#2

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
t
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.listOfIndeividualsAndTheirFitness = list()
        self.rankedListOfIndeividualsAndTheirFitness = list()
        self.roulletWheelOfIndeividualsAndTheirFitness = list()
        self.setOfProblem = setOfProblem
        self.creatingPopulation(self.setOfProblem)
        self.breeding()
        self.resultIndex = self.rankedListOfIndeividualsAndTheirFitness[0].genome
```



```

self.resultFitness = self.rankedListOfIndeividualsAndTheirFitness[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.listOfIndeividualsAndTheirFitness.append(Individual(genome,
self.forEvaluation(genome, setOfProblem)))
self.rankOfEachIndeividualByMeansOfRouletteWheelApproach()
self.rankedAsRouletteWheel()
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 rankOfEachIndeividualByMeansOfRouletteWheelApproach(self):
self.rankedListOfIndeividualsAndTheirFitness =
sorted(self.listOfIndeividualsAndTheirFitness,
key=cmp_to_key(self.compare),reverse = True)
def rankedAsRouletteWheel(self):
length = len(self.rankedListOfIndeividualsAndTheirFitness)
for eachIndeividual in
range(len(self.rankedListOfIndeividualsAndTheirFitness)):
genome, _ = self.rankedListOfIndeividualsAndTheirFitness[eachIndeividual]
length-=1
self.rouletteWheelOfIndeividualsAndTheirFitness.append(Individual(genome, leng
th))
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.rankedAsRouletteWheel()
            self.rankedListOfIndevidualsAndTheirFitness =
            self.rankedListOfIndevidualsAndTheirFitness[:self.populationSize]
            self.rouletteWheelOfIndevidualsAndTheirFitness =
            self.rouletteWheelOfIndevidualsAndTheirFitness[:self.populationSize]
            def tournament(self, tournament_size=2):
                return max(random.choices(self.rouletteWheelOfIndevidualsAndTheirFitness,
                    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)

```

Written review for lab#2

peer review, lab#2 #4



msdkarimi opened this issue on Nov 13, 2022 · 0 comments



msdkarimi commented on Nov 13, 2022



Hi dear Juan,
I hope you're doing well.

Firstly, I have to confess that I learnt something new in terms of python coding in your codes.

Secondly, I did not get your point by "Number of Elements" and "Iterations" for founded solution, as far as I understood we are interested in counting the total number of digits(weight) in the covered solution. For instance, if N=100, the lowest weight that we can find is equal to 100, while in your read me file these values are 8 and 44 respectively.

From my point of view, it would be better if you checked whether your solution has covered all the range 0 to N-1.

In the function called "random_population" you try to create list of individuals, the way I understood your code is that each individual is just a list in your problem state, which is wrong way of representing the problem. In my opinion, each individual could be sublists of initial problem state or a mask of it in a way that "True" means we are supposed to pick that sublist and "False" means we are not interested in that sublist.

regards,
MSD

Received review for lab#2

lab2 review #4



shadow036 opened this issue on Nov 13, 2022 · 0 comments



shadow036 commented on Nov 13, 2022



Hello, I'm gonna do a quick review of the second lab, explaining in my opinion the parts which could have been implemented better and the things that I've found interesting.

1. Before reviewing the code, I'd like to give you an advice about the github repository organization, because it seems that you have a lot of files and folders which are not useful for reviewing.
In this case I think that if you add a .gitignore file in the root folder (together with lab01/ and lab2/) and write on it the following lines you should be OK also for the next labs:

```
.DS_Store
*/.idea/*
*/.DS_Store/*
**/tests/*
**/.DS_Store/*
**/pyproject.toml
```
2. First of all I think it would have been useful to add some sort of explanations regarding the various steps and functions in the README other than the results, since in my opinion the code you included is not so immediate to understand.
In this way you could have also avoided to use such long names for the functions.
3. You should have used `Ns` as `populationSize` instead of creating another list with different values.
4. Also in this case I don't think that it is necessary to use a class and to put everything (variables, functions, ...) inside of it because then it becomes really messy (but this is a just personal preference).
5. In the `creatingPopulation` block you immediately generate the initial individuals with a fixed (`problemSize`) number of sublists and based on the following mutations, this number remains the same throughout the whole process.
I don't think this is very good since you are considering only solutions with `problemSize` elements and if the optimal one isn't in there you'll never reach it.
Also for this reason I would have considered other options for mutations (in addition to the one that you used) such as adding a new sublist of removing one already present (in order to backtrack and increase the possibility of finding new individuals in future generations).
6. I don't think that is really necessary to compute the number of digits of the fitness function, but this is negligible.
7. Finally I liked the way that you tried to favor the evolution towards better individuals through the "roulette strategy", generating in this way a system which doesn't rely only on pure fitness and also at the same time standardizing the difference between each couple of following (based on fitness) individuals.

Overall I think this was a quite good job even though there were some major problems that either drastically limited the possibilities of finding a good solution [point 3] (especially for higher `Ns` but also for `N = 20` is noticeable) or set up the wrong basis for the problem representation [point 5], which then propagated throughout the whole code, making it not so relevant for the context.

Most of the problems came from those two point, while the others are mainly due to personal preference or can be simply ignored in this situation.

Lab#3

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["longests"], random.randint(1,
self._rows[self.inferedStatus["longests"]]))
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)+ ">"

```

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))

```

Received review for lab#3

Peer review lab 3 #5



gabrjc opened this issue last month · 0 comments



gabrjc commented last month



General Consideration:

For future work I recommend using Python Notebook to make the reading and analysis of your code easier.
Even a more depth Readme would make it easier to review your code :)

Task Consideration:

The "DoNimming" function return an error when performed in the main, but analyzing only the functions it seems that the function of GA is valid, as well as the random ones and the one based on Nimsum.
I don't understand the need to have three different nimming functions, maybe there was an easier way to do nimming :)

Your MinMax function is difficult to understand for me, but if I understand correctly I think it was necessary to build a tree, and choose from the possible_actions in a recursive way and not only stop at the first level.

Reinforcement Learning is missing

Final Project – Quarto

main.py

```
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.learnModelParameters(copy.deepcopy(game.availablePieces))
    # save("weightOfPlaceFinalExam.npy", placeReward)
    # save("weightOfPieceFinalExam.npy", pieceReward)

    # pieceReward = numpy.load('weightOfPieceExam.npy',
allow_pickle=True)
    # pieceReward = dict(enumerate(pieceReward.flatten(), 1))
```

```

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

# -----/\ Battle
Field /\-----
rounds = 1000
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.gainPiece = pieceReward
        agentRL.gainPlace = 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 random 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()

```

objects.py

```
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.moveHistory = [] # place, reward
        self.pieceHistory = [] # piece, reward
        self.alpha = alpha
        self.randomFactor = randomFactor
        self.gainPlace = {}
        self.gainPiece = {}
        self.learningPhase = learningPhase
        self.initReward(self.__quarto)
        self.rounds = rounds
        self.currentRound = 0
        self.tempSelected = set()

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

            for pI, _ in enumerate(quarto.get_all_pieces()):
                self.gainPiece[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.calculateWeight(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.gainPiece[new_piece_index] >= maxG or tempW >=
maxGrule:

                next_piece_index = new_piece_index
                maxG = self.gainPiece[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.gainPiece[new_piece_index] >= maxG:
            next_piece_index = new_piece_index
            maxG = self.gainPiece[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.updateSelectedPiecesAndPlaces()
    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.zippedFreePlaces]
            next_move = random.choice(freePLaces)
        else:
            pairXYfeasibility = []
            for placeXY in self.__quarto.zippedFreePlaces:
                weight = self.__quarto.calculateWeightPlace(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.gainPlace[new_state] >= maxG and palceWeight
>= maxGrun:
            next_move = new_state
            maxG = self.gainPlace[new_state]
            maxGrun = palceWeight

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

    else:
        for action in self.__quarto.zippedFreePlaces:
            new_state = action
            if self.gainPlace[new_state] >= maxG:
                next_move = new_state
                maxG = self.gainPlace[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.moveHistory.append((place, reward))

def updateMovesHistory(self, place):
    reward = self.__quarto.assignReward()
    self.moveHistory.append((place, reward))

def updatePieceHistory(self, piece):
    reward = self.__quarto.assignReward()
    self.pieceHistory.append((piece, reward))

```

```

class PropertyCounter(object):
    def __init__(self, selectedPieceCharacteristic = None):
        if selectedPieceCharacteristic == None:
            self.h = 0
            self.nh = 0
            self.c = 0
            self.nc = 0
            self.so = 0
            self.nso = 0
            self.sq = 0
            self.nsq = 0
        else:
            if selectedPieceCharacteristic.HIGH:
                self.h = 1
                self.nh = 0
            else:
                self.nh = 1
                self.h = 0

            if selectedPieceCharacteristic.COLOURED:
                self.c = 1
                self.nc = 0
            else:
                self.nc = 1
                self.c = 0

            if selectedPieceCharacteristic.SOLID:
                self.so = 1
                self.nso = 0
            else:
                self.nso = 1
                self.so = 0

            if selectedPieceCharacteristic.SQUARE:
                self.sq = 1
                self.nsq = 0
            else:
                self.nsq = 1
                self.sq = 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.zippedFreePlaces = None

```

```

        self.availablePieces = None

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

    def updateSelectedPiecesAndPlaces(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.zippedFreePlaces = zip(self.freePlaces[0], self.freePlaces[1])

    def calculateWeight(self, pieceIndex):
        badPrise = -10e10
        copyOfBoard = copy.deepcopy(self.__board)
        self.updateSelectedPiecesAndPlaces()
        thisFreePlacesRewardForGivenPiece = []
        for possition in self.zippedFreePlaces:
            x, y = possition
            self.__board = copy.deepcopy(copyOfBoard)
            self.__board[x, y] = pieceIndex
            _, dictH = self.__check_horizontal()
            _, dictV = self.__check_vertical()
            _, dictD = self.my_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.h, propertyH.nh, propertyH.c, propertyH.nc,
propertyH.so, propertyH.nso, propertyH.sq,
                    propertyH.nsq])
                thisStepV = numpy.array(

```

```

        [propertyV.h, propertyV.nh, propertyV.c, propertyV.nc,
propertyV.so, propertyV.nso, propertyV.sq,
        propertyV.nsq])
        l1 = dictLTR[0]
        l2 = dictLTR[1]
        l3 = dictLTR[2]
        l4 = dictLTR[3]
        thisStepL = numpy.array([l1.h + l2.h + l3.h + l4.h, l1.nh
+ l2.nh + l3.nh + l4.nh,
                                l1.c + l2.c + l3.c + l4.c, l1.nc
+ l2.nc + l3.nc + l4.nc,
                                l1.so + l2.so + l3.so + l4.so,
l1.nso + l2.nso + l3.nso + l4.nso,
                                l1.sq + l2.sq + l3.sq + l4.sq,
l1.nsq + l2.nsq + l3.nsq + l4.nsq])

        r1 = dictRTL[0]
        r2 = dictRTL[1]
        r3 = dictRTL[2]
        r4 = dictRTL[3]
        thisStepR = numpy.array([r1.h + r2.h + r3.h + r4.h, r1.nh
+ r2.nh + r3.nh + r4.nh,
                                r1.c + r2.c + r3.c + r4.c, r1.nc
+ r2.nc + r3.nc + r4.nc,
                                r1.so + r2.so + r3.so + r4.so,
r1.nso + r2.nso + r3.nso + r4.nso,
                                r1.sq + r2.sq + r3.sq + r4.sq,
r1.nsq + r2.nsq + r3.nsq + r4.nsq])

        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 calculateWeightPlace(self, pieceIndex, placeXY):
        grandPrise = 10e10
        worstPrise = -grandPrise
        # 2 file 5
        thresholdFor3 = 2
        copyOfBoard = self.get_board_status()

```

```

self.updateSelectedPiecesAndPlaces()
thisFreePlacesRewardForGivenPiece = []

x, y = placeXY
self.__board = copy.deepcopy(copyOfBoard)
self.__board[x, y] = pieceIndex
_, dictH = self.__check_horizontal()
_, dictV = self.__check_vertical()
_, dictD = self.my_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.h, propertyH.nh, propertyH.c, propertyH.nc,
propertyH.so, propertyH.nso, propertyH.sq,
propertyH.nsq])
    thisStepV = numpy.array(
        [propertyV.h, propertyV.nh, propertyV.c, propertyV.nc,
propertyV.so, propertyV.nso, propertyV.sq,
propertyV.nsq])
    l1 = dictLTR[0]
    l2 = dictLTR[1]
    l3 = dictLTR[2]
    l4 = dictLTR[3]
    thisStepL = numpy.array([l1.h + l2.h + l3.h + l4.h, l1.nh +
l2.nh + l3.nh + l4.nh,
                                l1.c + l2.c + l3.c + l4.c, l1.nc +
l2.nc + l3.nc + l4.nc,
                                l1.so + l2.so + l3.so + l4.so, l1.nso
+ l2.nso + l3.nso + l4.nso,
                                l1.sq + l2.sq + l3.sq + l4.sq, l1.nsq
+ l2.nsq + l3.nsq + l4.nsq])

    r1 = dictRTL[0]
    r2 = dictRTL[1]
    r3 = dictRTL[2]
    r4 = dictRTL[3]
    thisStepR = numpy.array([r1.h + r2.h + r3.h + r4.h, r1.nh +
r2.nh + r3.nh + r4.nh,
                                r1.c + r2.c + r3.c + r4.c, r1.nc +
r2.nc + r3.nc + r4.nc,

```

```

r1.so + r2.so + r3.so + r4.so, r1.nso
+ r2.nso + r3.nso + r4.nso,
r1.sq + r2.sq + r3.sq + r4.sq, r1.nsq
+ r2.nsq + r3.nsq + r4.nsq])

```

```

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 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_characteristics(self, index: int) -> Piece:
    """
    Gets characteristics 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:
    horDict = dict()

```

```

        horDict[0] = None
        horDict[1] = None
        horDict[2] = None
        horDict[3] = None

    for i in range(self.BOARD_SIDE):
        initiallist = PropertyCounter()

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

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

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

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

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

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

```

```

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

        circle_values = [
            elem for elem in self.__board[i] if elem >= 0 and not
self.__pieces[elem].SQUARE
        ]
        initiallist.nsq = len(circle_values)
        horDict[i] = initiallist

        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, horDict

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

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

```

```

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

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

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

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

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

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

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

        verDict[i] = initiallist

```

```

        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, verDict

```

```

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 = PropertyCounter()

```

```

        if self.__pieces[self.__board[i, i]].HIGH:

```

```

            if self.__board[i, i] != -1:
                high_values.append(self.__board[i, i])
                LTRinitiallist.h = len(high_values)

```

```

        else:

```

```

            if self.__board[i, i] != -1:
                low_values.append(self.__board[i, i])
                LTRinitiallist.nh = 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.c = len(coloured_values)
        else:
            if self.__board[i, i] != -1:
                noncolor_values.append(self.__board[i, i])
                LTRinitiallist.nc = 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.so = len(solid_values)
        else:
            if self.__board[i, i] != -1:
                hollow_values.append(self.__board[i, i])
                LTRinitiallist.nso = 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.sq = len(square_values)
        else:
            if self.__board[i, i] != -1:
                circle_values.append(self.__board[i, i])
                LTRinitiallist.nsq = 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 = PropertyCounter()
    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.h = 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.nh = 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.c = 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.nc = 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.so = 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.nso = 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.sq = 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.nsq = 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 my_check_diagonal(self):
    LTRdiagDict = dict()
    LTRdiagDict[0] = None
    LTRdiagDict[1] = None
    LTRdiagDict[2] = None

```

```
LTRdiagDict[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 = PropertyCounter()

    if self.__pieces[self.__board[i, i]].HIGH:
        if self.__board[i, i] != -1:
            high_values.append(self.__board[i, i])
            LTRinitiallist.h = len(high_values)
        else:
            if self.__board[i, i] != -1:
                low_values.append(self.__board[i, i])
                LTRinitiallist.nh = 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.c = len(coloured_values)
        else:
            if self.__board[i, i] != -1:
                noncolor_values.append(self.__board[i, i])
                LTRinitiallist.nc = 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.so = len(solid_values)
        else:
            if self.__board[i, i] != -1:
                hollow_values.append(self.__board[i, i])
                LTRinitiallist.nso = 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.sq = len(square_values)
        else:
            if self.__board[i, i] != -1:
                circle_values.append(self.__board[i, i])
                LTRinitiallist.nsq = 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

    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 = PropertyCounter()
        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.h = 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.nh = 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.c = 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.nc = 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.so = 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.nso = 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.sq = 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.nsq = 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 learnModelParameters(self, pieces):
    agentRL = 0
    agentRandom = 0
    draw = 0
    if self.__players[0].learningPhase:
        print("inlearning ")

    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.updateSelectedPiecesAndPlaces()
            piece_ok = False
            while not piece_ok:
                self.updateSelectedPiecesAndPlaces()
                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].updatePieceHistory(self.__selected_piece_index)
                piece_ok = False
                self.__current_player = (self.__current_player + 1) %
self.MAX_PLAYERS

                while not piece_ok:

```

```

        self.updateSelectedPiecesAndPlaces()
        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].updateMovesHistory(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].moveHistory = []
            self.__players[0].pieceHistory = []

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

def learn(self, player):
    target = 0
    for prev, reward in reversed(player.moveHistory):
        player.gainPlace[prev] = player.gainPlace[prev] + player.alpha
* (target - player.gainPlace[prev])
        target += reward

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

```

```

        player.moveHistory= []
        player.pieceHistory = []

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

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

```