COMPUTATIONAL INTELLIGENCE REPORT (2022-2023) (14-06-2023)

Name	**Mohamed Khaled Hassan Aly**	**GitHub Account**	https://github.com/MoMido1
Surname	**MOTRASH**	**Repository**	https://github.com/MoMido1/Computational_Intellegence_2022
Email	S295805@studenti.polito.it	**Exam Date**	14-06-2023

TOTAL WORK THROUGH THE YEAR

LABS	Details	Details	Contributors
Lab1 **Set Covering Problem**	Breadth First	Developed without any external resources	**Alone**
	Depth First	Developed without any external resources	**Alone**
	Greedy BF	Developed without any external resources	**Alone**
	A*	Developed without any external resources	**Alone**
	Link	https://github.com/MoMido1/Computational_Intellegence_2022/tree/main/Lab1	
Lab1_HillClimbing	Description	Solving the same problem of set covering using the same problem function with the same available function.	
	Tweak	The tweak function removes an already covered element and places a new one that is not covered and searches for the new solution if there's any	**Alone**
	Type	2 types of Hill Climbing were attempted: - SS -> Steepest-step FI -> First improvement	**Alone**
	All this code was developed without any external resource or assistance		
	Link	https://github.com/MoMido1/Computational_Intellegence_2022/tree/main/Lab1_Hill_Climping	
Lab2_setCovering+ea **Evolutionary Algorithms**	Description	Solving the set covering problem using the same problem specifications with Evolutionary Algorithms. No external resources were used	
	Types	Different types of Algorithms were used: - (1 + 1) Algorithm (1 + λ) Algorithm (1 , λ) Algorithm (μ , λ) Algorithm	**Alone**
	Link	https://github.com/MoMido1/Computational_Intellegence_2022/tree/main/lab2_setCovering%2Bea	
GA Set Covering Problem **Genetic Algorithm**	Details	Applying genetic Algorithm for set covering problem and modifying the hyper parameters	
	Types of Operators	Several operators were used with different probability of selection in each generation: 1- Cross Over i.e **(prob = 0.1)** 2- Mutation i.e **(prob = 0.1)** 3- Elitism i.e **(prob = 0.8)** **(ALONE)**	**Inspired from Lecture Code**
	Parent Selection	Tournament selection (pressure = 2) Random parent selection	**Inspired from Lecture Code**
	Link	https://github.com/MoMido1/Computational_Intellegence_2022/tree/main/GA_SetCovering_Problem	
Lab3_NIM_Game	Details	4 tasks were attempted to play against random player	
	Expert Player	Using the Nim sum technique to decide	**Inspired from Lecture Code**

Base-Nim	Here it makes a check based on a base-3 NIM sum	**Alone**
Min Max	Uses the min max strategy to find the best solution	**Inspired from Lecture Code**
Reinforcement Leaning	Using Markov Decision Process algorithm for learning	**Inspired from online code presented in lecture**
Link	https://github.com/MoMido1/Computational_Intellegence_2022/tree/main/Lab3_NIM_Game	

PROJECT (2 ALGORITHMS WERE ATTEMPTED)

Quarto Game	details	Contribution
Expert Player	Places the piece in the best place possible to guarantee the win and if not, it will place it in a place to make the opponent not win in any case. Then selects the piece for the opponent that won't guarantee for the opponent the win	**Alone**
Minmax Player	Uses Min Max algorithm for better selection of the piece and a better selection of the position.	**Alone**
Link	https://github.com/MoMido1/Computational_Intellegence_2022/tree/main/quarto	

LAB1 CODE

```
import random
import collections
import networkx as nx
import matplotlib.pyplot as plt
```

In [2]:

```
def problem(N, seed=None):

    random.seed(seed)
    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))
    ]
    #the rank here could be considered as the cost
    ranked_list = [(x,len(x)) for x in array]

    return ranked_list
```

In [3]:

```
def Breadth_First (N,lst):
#Breadth first aproach
    Nums = list(range(N))
    Covered_Nums = list()
    selected_Lists = list()
    weight=int()


    arranged_list = sorted(lst, key= lambda x: x[1], reverse= False)
    # the arranged_list arranges the nodes increasingly from lists with single element to more
complex lists


    f=0
    for e in arranged_list:
        for i in e[0]:
            if i in Covered_Nums:
                continue
            else:
                Covered_Nums.append(i)
                f=1
        if f==1 :
            selected_Lists.append(e[0])
            f=0
        if sorted(collections.Counter(Nums)) == sorted(collections.Counter(Covered_Nums)):
            break

    weight = sum(len(x) for x in selected_Lists)
    print(f"For N = {N} => the list has a weight W = {weight} and we used {len(selected_Lists)}
nodes to cover it")
```

In [4]:

```
def depthFirst (N,lst):
    #this is a depth first aproach
    Nums = list(range(N))
    Covered_Nums = list()
    selected_Lists = list()
    weight=int()

    # arranged_list = sorted(list(ranked_list), key= lambda x: x[1], reverse= False)
    depthFirst_list = sorted(lst, key= lambda x: x[1], reverse= True)
    # print(rev_arranged_list)
```

```
    # depthFirst_list = rev_arranged_list

    # maxrank =arranged_list[-1][1]
    # # we want to get the max number of rank that exists
    # depthFirst_list = list()
    # #creating the depth first list
    # while len(arranged_list) :
    #     for i in range(maxrank+1):
    #         for j in range(len(arranged_list)):
    #             if arranged_list[j][1] == i:
    #                 depthFirst_list.append(arranged_list[j])
    #                 arranged_list.pop(j)
    #                 break
    # print(depthFirst_list)
    f=0
    for e in depthFirst_list:
        for i in e[0]:
            if i in Covered_Nums:
                continue
            else:
                Covered_Nums.append(i)
                f=1
        if f==1 :
            selected_Lists.append(e[0])
            f=0
        if sorted(collections.Counter(Nums)) == sorted(collections.Counter(Covered_Nums)):
            # here i want to compare two lists and i can't do that in a list so i converted it
to a collection ans used the
            #counter method to count regardless of the position of the elements and that sorts
the elements alphabitacally
            break

    weight = sum(len(x) for x in selected_Lists)
    print(f"For N = {N} => the list has a weight W = {weight} and we used {len(selected_Lists)}
nodes to cover it")
```

In [5]:

```
def greedyBestFirst (N,lst):
    #this is a greedy best first aproach
    Nums = list(range(N))
    Covered_Nums = list()
    selected_Lists = list()
    weight=int()

    arranged_list = sorted(lst, key= lambda x: x[1], reverse= True)

    while collections.Counter(Nums) != collections.Counter(Covered_Nums):
        f=0
        e = arranged_list[0]
        for i in e[0]:
            if i in Covered_Nums:
                continue
            else:
                Covered_Nums.append(i)
                f=1
        if f==1 :
            selected_Lists.append(e[0])
            f=0
        if sorted(collections.Counter(Nums)) == sorted(collections.Counter(Covered_Nums)):
            #we found a solution
```

```
            break

        for i in range(len(arranged_list)):
            #changing the tuple to list to adjust it
            element = list(arranged_list[i])
            # in the previous methods we assumed that the length of the list represents the
priority
            # but here we will change the priority place and we will put a new priority that
will be equal to
            # the number of new numbers that are not covered in our set
            element[1] = len(list(set(arranged_list[i][0]).difference(Covered_Nums)))
            # so each element will have different priority each time as we will give the
element with highest number of uncovered
            #numbers the highest priority
            arranged_list[i]= tuple(element) # now we replace that element with the new one
with the new priotiy
        arranged_list = sorted(arranged_list, key= lambda x: x[1], reverse= True)
        # after modifying the priorities we then rearrange the list to know the new element to
select
        #and the next cycle we will select only the first element and then recalculating

    weight = sum(len(x) for x in selected_Lists)
    print(f"For N = {N} => the list has a weight W = {weight} and we used {len(selected_Lists)}
nodes to cover it")
```

In [6]:

```
#This is A* strategy where the function h() will be the same as we used in the greedy best
first and now the actual cost
#will be the cost of selecting a node which has a high priority but very large numbers over all
so it increases the weight
# so now we make that the function will calculate a cost which is when the number of new
numbers is greater that 75% of the numbers
def A_ (N,lst):
    #this is a A* first aproach
    Nums = list(range(N))
    Covered_Nums = list()
    selected_Lists = list()
    weight=int()

    arranged_list = sorted(lst, key= lambda x: x[1], reverse= True)

    while collections.Counter(Nums) != collections.Counter(Covered_Nums):
        f=0
        e = arranged_list[0]
        for i in e[0]:
            if i in Covered_Nums:
                continue
            else:
                Covered_Nums.append(i)
                f=1
        if f==1 :
            selected_Lists.append(e[0])
            f=0
        if sorted(collections.Counter(Nums)) == sorted(collections.Counter(Covered_Nums)):
            #we found a solution
            break

        for i in range(len(arranged_list)):
            #changing the tuple to list to adjust it
            element = list(arranged_list[i])
```

```
            newNumbers = len(list(set(arranged_list[i][0]).difference(Covered_Nums)))
            oldNumbers = len(element[0])- newNumbers
            priority = newNumbers - (oldNumbers/len(element[0])) * newNumbers
            # we make a penalty of a percentatage of the old numbers compared to the new
numberes to reduce the priority of the element
            element[1] = priority
            arranged_list[i]= tuple(element) # now we replace that element with the new one
with the new priotiy
        arranged_list = sorted(arranged_list, key= lambda x: x[1], reverse= True)


    weight = sum(len(x) for x in selected_Lists)
    print(f"For N = {N} => the list has a weight W = {weight} and we used {len(selected_Lists)}
nodes to cover it")
```

In [7]:

```
print("---------Breadth first Algorithm------------")
for N in [5, 10, 20, 100, 500, 1000]:
    sspace = nx.Graph()
    nodes = problem(N,42)
    for node_list in nodes:
        sspace.add_node(tuple(node_list[0]))
    # sspace.add_nodes_from(nodes)

    Breadth_First(N,nodes)

    # plt.figure(figsize=(12, 8))
    # nx.draw(sspace, with_labels=True)

print("\n---------Depth first Algorithm------------")
for N in [5, 10, 20, 100, 500, 1000]:
    depthFirst(N,problem(N,42))

print("\n---------Greedy best-first Algorithm------------")
for N in [5, 10, 20, 100, 500, 1000]:
    greedyBestFirst(N,problem(N,42))

print("\n------- A* Algorithm ------------")
for N in [5, 10, 20, 100, 500, 1000]:
    A_(N,problem(N,42))
```

```
---------Breadth first Algorithm------------
For N = 5 => the list has a weight W = 5 and we used 5 nodes to cover it
For N = 10 => the list has a weight W = 13 and we used 7 nodes to cover it
For N = 20 => the list has a weight W = 46 and we used 12 nodes to cover it
For N = 100 => the list has a weight W = 332 and we used 19 nodes to cover it
For N = 500 => the list has a weight W = 2162 and we used 24 nodes to cover it
For N = 1000 => the list has a weight W = 4652 and we used 26 nodes to cover it

---------Depth first Algorithm------------
For N = 5 => the list has a weight W = 8 and we used 4 nodes to cover it
For N = 10 => the list has a weight W = 19 and we used 4 nodes to cover it
For N = 20 => the list has a weight W = 57 and we used 7 nodes to cover it
For N = 100 => the list has a weight W = 379 and we used 9 nodes to cover it
For N = 500 => the list has a weight W = 2044 and we used 10 nodes to cover it
For N = 1000 => the list has a weight W = 5242 and we used 13 nodes to cover it

---------Greedy best-first Algorithm------------
For N = 5 => the list has a weight W = 6 and we used 3 nodes to cover it
For N = 10 => the list has a weight W = 13 and we used 3 nodes to cover it
```

```
For N = 20 => the list has a weight W = 32 and we used 4 nodes to cover it
For N = 100 => the list has a weight W = 191 and we used 5 nodes to cover it
For N = 500 => the list has a weight W = 1375 and we used 7 nodes to cover it
For N = 1000 => the list has a weight W = 3087 and we used 8 nodes to cover it

------- A* Algorithm ------------
For N = 5 => the list has a weight W = 5 and we used 3 nodes to cover it
For N = 10 => the list has a weight W = 10 and we used 3 nodes to cover it
For N = 20 => the list has a weight W = 28 and we used 4 nodes to cover it
For N = 100 => the list has a weight W = 166 and we used 5 nodes to cover it
For N = 500 => the list has a weight W = 1297 and we used 8 nodes to cover it
For N = 1000 => the list has a weight W = 2776 and we used 8 nodes to cover it
```

Lab2- setCovering Problem with HillClimping Techniques
proceeding with the set covering problem but using Hill Climbing Techniques:

In [201]:

```
import random
import logging
import collections
```

In [202]:

```
def problem(N, seed=None):
    random.seed(seed)
    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))
    ]
    ranked_list = [(sorted(x),len(x)) for x in array]
    return ranked_list
```

1- Hill Climping: -

In [203]:

```
def better(old,new,discovered_Nums):
    if len(new)< len(old):
        # w is less for sure so we select the new node
        return True
    elif len(new) >= len(old):
        newCover = len(collections.Counter(new) - collections.Counter(discovered_Nums))
        oldCover = len(collections.Counter(old) - collections.Counter(discovered_Nums))
        if newCover > oldCover:
            return True
        else:
            return False
    else:
        return False

def bestNode(betterNodes,discovered_Nums):
    best= betterNodes[0]
    for i in range(len(betterNodes)):
        if len(best)>= len(betterNodes[i]):
            bestCover = len(collections.Counter(best) - collections.Counter(discovered_Nums))
            nodeCover = len(collections.Counter(betterNodes[i]) -
collections.Counter(discovered_Nums))
            if nodeCover >= bestCover:
                best = betterNodes[i]
        else:
            bestCover = len(collections.Counter(best) - collections.Counter(discovered_Nums))
            nodeCover = len(collections.Counter(betterNodes[i]) -
collections.Counter(discovered_Nums))
            if nodeCover == bestCover:
                best = betterNodes[i]
    return best
```

In [207]:

```
def Tweak(state,nodes,discovered_Nums,N,mtype="fi"):
    # here i'm trying to select the perfect state to add and update the discovered_Nums so
until now we are exploiting
    stateNewNums = collections.Counter(state) - collections.Counter(discovered_Nums)
    # the stateNewNums are the nums that are not in the discovered nums yet
    stateOldNums = list((collections.Counter(state)- stateNewNums).keys())
    initstateOldNums = len(stateOldNums)
    nwState = state.copy()
```

```
    Nums = list(range(N))
    remNumstoCover_notAlready_Instate =list((collections.Counter(Nums) -
collections.Counter(discovered_Nums)- collections.Counter(state)).keys())
    if len(remNumstoCover_notAlready_Instate)==0:
        return state
    rn = random.choice(remNumstoCover_notAlready_Instate)
    flag =1
    numtoRm= int()
    betterNodes = list()
    bestOfNodes = list()
    firstSolution= list()

    while len(remNumstoCover_notAlready_Instate) != 0:
        if initstateOldNums == 0 :
            # so here all the state numbers are new
            nwState.append(rn)
            if tuple([sorted(nwState),len(nwState)]) in nodes:
                if (better(state,nwState,discovered_Nums)) & (mtype == "ss"):
                    # add to betterNodes
                    betterNodes.append(nwState)
                    nwState= state.copy()
                    remNumstoCover_notAlready_Instate.remove(rn)
                    if len(remNumstoCover_notAlready_Instate) == 0:
                    # here we didn't find any neighbour
                        break
                    rn =random.choice(remNumstoCover_notAlready_Instate)
                    continue

                elif (better(state,nwState,discovered_Nums)) & (mtype == "fi"):
                    #add to the first solution and return it
                    firstSolution = nwState
                    break

            else:
                nwState= state.copy()
                remNumstoCover_notAlready_Instate.remove(rn)
                if len(remNumstoCover_notAlready_Instate) == 0:
                    # here we didn't find any neighbour
                    break
                rn =random.choice(remNumstoCover_notAlready_Instate)
        else:
            if flag:
                numtoRm =random.choice(stateOldNums)
                stateOldNums.remove(numtoRm)
                nwState.remove(numtoRm)
                flag = 0

            nwState.append(rn)
            if tuple([sorted(nwState),len(nwState)]) in nodes:
                if (better(state,nwState,discovered_Nums))& (mtype == "ss"):
                    # add to betterNodes
                    betterNodes.append(nwState)
                    nwState= state.copy()
                    remNumstoCover_notAlready_Instate.remove(rn)
                    if len(remNumstoCover_notAlready_Instate) == 0:
                    # here we didn't find any neighbour that exist in our search problem
                        break
                    rn =random.choice(remNumstoCover_notAlready_Instate)
                    continue
```

```
                elif (better(state,nwState,discovered_Nums))& (mtype == "fi"):
                    #add to the first solution and return it
                    firstSolution = nwState
                    break

            else:
                nwState.remove(rn)
                remNumstoCover_notAlready_Instate.remove(rn)
                if len(remNumstoCover_notAlready_Instate) == 0:
                    flag =1
                    remNumstoCover_notAlready_Instate =list((collections.Counter(Nums) -
collections.Counter(discovered_Nums)- collections.Counter(state)).keys())
                    if len(stateOldNums) == 0 :
                        break
                rn =random.choice(remNumstoCover_notAlready_Instate)

    if len(betterNodes) !=0:
        bestOfNodes= bestNode(betterNodes,discovered_Nums)
    else:
        bestOfNodes = state

    if mtype=="fi":
        return firstSolution if len(firstSolution)!=0 else state
    else:
        return bestOfNodes
```

In [ ]:

```
N = 100
nodes = problem(N,42)
print(nodes)
Tweak(state=[8,2,7,6],nodes=nodes,discovered_Nums=[8,2,6],N=N)
```

In [210]:

```
# first improvement => fi
# Steepest-step => ss
def Hill_Climping_Search(nodes,N,seed =42,mtype ="ss"):
      random.seed(seed)
      discovered_Nums=list()
      st_Index = random.randint(0, len(nodes)-1)
      st_node = nodes[st_Index][0]
      solution_nodes = list()

      for _ in range(len(nodes)):
            nwnode= Tweak(state=st_node,nodes=nodes,
                  discovered_Nums=discovered_Nums,
                  N=N,
                  mtype=mtype)
            discovered_Nums.extend(list((collections.Counter(nwnode)-
collections.Counter(discovered_Nums)).keys()))
            solution_nodes.append(nwnode)

            if len(discovered_Nums) == N:
                  break

            nodes.remove(tuple([sorted(nwnode),len(nwnode)]))
            st_Index = random.randint(0, len(nodes)-1)
            st_node = nodes[st_Index][0]
      weight = sum(len(x) for x in solution_nodes)
      print(f"For N = {N} => the list has a weight W = {weight} and we used
{len(solution_nodes)} nodes to cover it")
```

In [211]:

```
print("---------Hill Climping Algorithm------------")
for N in [5, 10, 20, 100, 500, 1000]:
    nodes = problem(N,42)
    Hill_Climping_Search(nodes,N)
```

```
---------Hill Climping Algorithm------------
For N = 5 => the list has a weight W = 5 and we used 3 nodes to cover it
For N = 10 => the list has a weight W = 24 and we used 7 nodes to cover it
For N = 20 => the list has a weight W = 59 and we used 9 nodes to cover it
For N = 100 => the list has a weight W = 347 and we used 14 nodes to cover it
For N = 500 => the list has a weight W = 2991 and we used 20 nodes to cover it
For N = 1000 => the list has a weight W = 5797 and we used 19 nodes to cover it
```

Lab 2: Set Covering with EA

In [9]:

```
import random
import logging
import collections
import numpy as np
import math
```

In [10]:

```
def problem(N, seed=None):
    random.seed(seed)
    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))
    ]
    ranked_list = [(sorted(x),len(x)) for x in array]
    return ranked_list
```

In [11]:

```
def better(old,new,discovered_Nums):
    if len(new)< len(old):
        # w is less for sure so we select the new node
        return True
    elif len(new) >= len(old):
        newCover = len(collections.Counter(new) - collections.Counter(discovered_Nums))
        oldCover = len(collections.Counter(old) - collections.Counter(discovered_Nums))
        if newCover > oldCover:
            return True
        else:
            return False
    else:
        return False

def bestNode(betterNodes,discovered_Nums):
    best= betterNodes[0]
    for i in range(len(betterNodes)):
        if len(best)>= len(betterNodes[i]):
            bestCover = len(collections.Counter(best) - collections.Counter(discovered_Nums))
            nodeCover = len(collections.Counter(betterNodes[i]) -
collections.Counter(discovered_Nums))
            if nodeCover >= bestCover:
                best = betterNodes[i]
        else:
            bestCover = len(collections.Counter(best) - collections.Counter(discovered_Nums))
            nodeCover = len(collections.Counter(betterNodes[i]) -
collections.Counter(discovered_Nums))
            if nodeCover == bestCover:
                best = betterNodes[i]
    return best
```

In [12]:

```
def Tw_1p1(state,stateOldNums,N):
    sigma = 0.01 *N
    newState =[]
    newNums =[]
    if not stateOldNums:
        return sorted(state)
    state = list((collections.Counter(state) - collections.Counter(stateOldNums)).keys())
    while len(set(newState)) != (len(stateOldNums)+len(state)):
        # the while loop guarantees that for us we keep producing multiple generation until
```

```
        # we end up with a generation with the best predictable performace
        newNums = np.random.normal(loc=stateOldNums,scale=sigma,size=(len(stateOldNums),))
        newNums = [math.ceil(x) for x in newNums]
        newNums = np.clip(newNums, a_min=0, a_max=None)
        newState = state.copy()
        newState.extend(newNums)
    return sorted(newState)

def Tw_1plmda(state,stateOldNums,discovered_Nums,N):
    lmda = math.ceil(0.01 *N)
    bestOfNodes = state
    offspring = list()
    offspring.append(state)
    for _ in range(lmda):
        offspring.append(Tw_1p1(state,stateOldNums,N))

    if len(offspring) !=0:
        bestOfNodes= bestNode(offspring,discovered_Nums)
    else:
        bestOfNodes = state
    return bestOfNodes

def Tw_1clmda(state,stateOldNums,discovered_Nums,N):
    lmda = math.ceil(0.2 *N)
    bestOfNodes = state
    offspring = list()
    for _ in range(lmda):
        offspring.append(Tw_1p1(state,stateOldNums,N))

    if len(offspring) !=0:
        bestOfNodes= bestNode(offspring,discovered_Nums)
    else:
        bestOfNodes = state
    return bestOfNodes

def Tw_uclmda(nodes,discovered_Nums,N):
    u = math.ceil(0.2* N)
    lmda = math.ceil(0.4 *N)

    nodes= np.array(nodes,dtype = object)
    offSpring = np.array(best_u_of_Nodes(nodes,u,discovered_Nums))[:,0]
    newOffSpring=[]
    n = lmda // u
    for state in offSpring:
        stateNewNums = collections.Counter(state) - collections.Counter(discovered_Nums)
        stateOldNums = tuple((collections.Counter(state)- stateNewNums).keys())
        newOffSpring.append((state,len(stateNewNums.keys())))
        for _ in range(n):
            newstate =Tw_1p1(state,stateOldNums,N)
            nw= collections.Counter(newstate) - collections.Counter(discovered_Nums)
            old = collections.Counter(newstate) - nw
            fitness = len(nw) - len(old)
            newOffSpring.append((newstate,fitness))
    newOffSpring = sorted(newOffSpring,key=lambda x:x[1])

    return newOffSpring[0:u]
def best_u_of_Nodes(nodes,u,discoveredNumbers):
    for node in nodes:
        nw= collections.Counter(node[0]) - collections.Counter(discoveredNumbers)
        old = collections.Counter(node[0]) - nw
```

```
        node[1] = len(nw) - len(old)
    sorted(nodes,key=lambda x: x[1],reverse=False)
    return nodes[0:u]
```

In [13]:

```
def Tweak(state,nodes,discovered_Nums,N,mtype):
    stateNewNums = collections.Counter(state) - collections.Counter(discovered_Nums)
    stateOldNums = tuple((collections.Counter(state)- stateNewNums).keys())
    newState = state
    if (mtype == '1p1'):
        newState = Tw_1p1(state,stateOldNums,N)
        if not better(state,newState,discovered_Nums):
            newState= state
    elif (mtype == '1plmda'):
        newState = Tw_1plmda(state,stateOldNums,discovered_Nums,N)
    elif (mtype == '1clmda'):
        newState = Tw_1clmda(state,stateOldNums,discovered_Nums,N)
    elif (mtype == 'uclmda'):
        newState = Tw_uclmda(nodes,discovered_Nums,N)
    else:
        print('unvalid Algorithm')


    return newState
```

In [14]:

```
# "1p1" -> 1+1
# "1plmda" -> 1 + Lambda
# "1clmda" -> 1 , Lambda
# "uclmda" -> u , Lambda
def evolutionary_Strategy(nodes,N,seed =42,mtype ="1p1"):
      random.seed(seed)
      discovered_Nums=list()
      st_Index = random.randint(0, len(nodes)-1)
      st_node = nodes[st_Index][0]
      solution_nodes = list()

      if mtype == 'uclmda':
            for _ in range(len(nodes)):
                  nwnodes= Tweak(state=st_node,nodes=nodes,
                  discovered_Nums=discovered_Nums,
                  N=N,
                  mtype=mtype)
                  best_node = nwnodes[0][0]
                  for node in nwnodes:
                        if better(best_node,node[0],discovered_Nums):
                              best_node= node[0]
                  discovered_Nums.extend(list((collections.Counter(best_node)-
collections.Counter(discovered_Nums)).keys()))
                  solution_nodes.append(best_node)
                  if len(discovered_Nums) == N:
                        break
                  if not nodes:
                        print("Algorithm failed to find any solution")
                        return None
      else:

            for _ in range(len(nodes)):
                  nwnode= Tweak(state=st_node,nodes=nodes,
                        discovered_Nums=discovered_Nums,
```

```
                        N=N,
                        mtype=mtype)
                discovered_Nums.extend(list((collections.Counter(nwnode)-
collections.Counter(discovered_Nums)).keys()))
                solution_nodes.append(nwnode)

                if len(discovered_Nums) == N:
                        break

                nodes.remove(tuple([sorted(st_node),len(st_node)]))
                if not nodes:
                        print("Algorithm failed to find any solution")
                        return None
                st_Index = random.randint(0, len(nodes)-1)
                st_node = nodes[st_Index][0]
        weight = sum(len(x) for x in solution_nodes)
        print(f"For N = {N} => the list has a weight W = {weight} and we used
{len(solution_nodes)} nodes to cover it")
```

In [15]:

```
STRATEGY_TYPE = {'1 + 1': "1p1" , '1 + Lambda':"1plmda" ,'1 , Lambda':"1clmda",'u + Lambda':
"uclmda"}
POPULATION_SIZES = [5, 10, 20, 100]
```

In [16]:

```
print("---------Evolutionary Strategies------------")
print('---------- (1 + 1)-ES ----------')
for N in POPULATION_SIZES:
    nodes = problem(N,42)
    evolutionary_Strategy(nodes,N,mtype=STRATEGY_TYPE["1 + 1"])

print('---------- (1 + Lambda)-ES ----------')
for N in POPULATION_SIZES:
    nodes = problem(N,42)
    evolutionary_Strategy(nodes,N,mtype=STRATEGY_TYPE["1 + Lambda"])

print('---------- (1 , Lambda)-ES ----------')
for N in POPULATION_SIZES:
    nodes = problem(N,42)
    evolutionary_Strategy(nodes,N,mtype=STRATEGY_TYPE["1 , Lambda"])

print('---------- (u + Lambda)-ES ----------')
for N in POPULATION_SIZES:
    nodes = problem(N,42)
    evolutionary_Strategy(nodes,N,mtype=STRATEGY_TYPE["u + Lambda"])
```

```
---------Evolutionary Strategies------------
---------- (1 + 1)-ES ----------
For N = 5 => the list has a weight W = 9 and we used 8 nodes to cover it
For N = 10 => the list has a weight W = 33 and we used 10 nodes to cover it
For N = 20 => the list has a weight W = 35 and we used 6 nodes to cover it
For N = 100 => the list has a weight W = 314 and we used 11 nodes to cover it
---------- (1 + Lambda)-ES ----------
For N = 5 => the list has a weight W = 18 and we used 15 nodes to cover it
For N = 10 => the list has a weight W = 28 and we used 9 nodes to cover it
For N = 20 => the list has a weight W = 35 and we used 6 nodes to cover it
For N = 100 => the list has a weight W = 314 and we used 11 nodes to cover it
---------- (1 , Lambda)-ES ----------
For N = 5 => the list has a weight W = 10 and we used 9 nodes to cover it
```

```
For N = 10 => the list has a weight W = 28 and we used 9 nodes to cover it
For N = 20 => the list has a weight W = 40 and we used 7 nodes to cover it
For N = 100 => the list has a weight W = 190 and we used 7 nodes to cover it
---------- (u + Lambda)-ES ----------
For N = 5 => the list has a weight W = 25 and we used 25 nodes to cover it
For N = 10 => the list has a weight W = 100 and we used 50 nodes to cover it
For N = 20 => the list has a weight W = 176 and we used 34 nodes to cover it
For N = 100 => the list has a weight W = 11948 and we used 427 nodes to cover it
```

GA-SETCOVERING PROBLEM CODE

Genetic Algorithm for Set Covering Problem

In [798]:

```
import random
import logging
from collections import namedtuple
import numpy as np
import math
```

In [799]:

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

In [800]:

```
PROBLEM_SIZE = 1000
POPULATION_SIZE = random.randint(PROBLEM_SIZE, PROBLEM_SIZE * 5)
OFFSPRING_SIZE = math.ceil(0.2* POPULATION_SIZE)
# we decided to make the OFFSpring number to be a ratio of the number of population and not a
fixed number but a fixed ratio

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

In [801]:

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

def fitness(genome):
    return sum(genome)/len(genome)

def problem_Encoding(array,N):
    population = list()
    for l in array:
        genome = tuple(1 if x in l else 0 for x in range(N))
        population.append(Individual(genome,fitness(genome)))
    return population
```

In [802]:

```
arr = problem(PROBLEM_SIZE,42)
population = problem_Encoding(arr,PROBLEM_SIZE)
```

In [803]:

```
def Random_select_parent(population):
    t = random.choice(population)
    return t

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

def mutation(population):
    p = Random_select_parent(population)
    n = random.randint(0,len(p.genome)-1)
    m = list(p.genome)
    m[n] = 1- m[n]
    return Individual(tuple(m),fitness(m))

def cross_over(population,tournament_size=2):
    p1 = tournament(population,tournament_size)
    p2 = tournament(population,tournament_size)
```

```python
    cut = random.randint(0,len(p1.genome)-1)
    ng = p1.genome[:cut] + p2.genome[cut:]
    return Individual(ng,fitness(ng))

def Elitism (population,tournament_size=2):
    #preserving the best of populations champion that are 0.01 of the population
    sortedPopulation = sorted(population,key = lambda i:i.fitness,reverse = True)
    cuttingInd = math.ceil(0.01*len(sortedPopulation))
    remainingPop = population[cuttingInd:]
    return cross_over(remainingPop)

#different types of genetic operators
GO = [mutation,cross_over,Elitism]
PROB = [0.1,0.1,0.8]

def get_one_Genetic_Operator():
    random_GO = random.choices(GO, PROB)[0]
    return random_GO
```

In [804]:

```python
# now we need to select the starting point
# this is selecting a specific individuals as parents
offSprings = list()
generations =0
for j in range(1_000_000):
    generations+=1
    #this problem could be solved if we make high iterations
    for i in range(OFFSPRING_SIZE):
        selectedGO =get_one_Genetic_Operator()

        o = selectedGO(population)
        offSprings.append(o)

    population.extend(offSprings)
    population = sorted(population,key = lambda i:i.fitness,reverse = True)
    population = population[:POPULATION_SIZE]
    offSprings.clear()
    if (population[0].fitness == 1):
        logging.info(f"found solution after {generations} generation")
        break
    if (j == 99999):
        logging.info(f"solution not found in 1_000_000 generation your Algorithm is poor")
```

```
INFO:root:found solution after 1238 generation
```

In [805]:

```python
population[0].fitness
```

Out[805]:

```
1.0
```

LAB3_NIM_GAME CODE

LAB3 POLICY Search

Task

Write agents able to play [*Nim*](https://en.wikipedia.org/wiki/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

```python
import logging
from itertools import permutations
from collections import namedtuple
import random

from copy import deepcopy,copy
from functools import reduce
import numpy as np
Nimply = namedtuple("Nimply", "row, num_objects")
class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self.num_rows = num_rows
        self._rows = [i * 2 + 1 for i in range(num_rows)] # here we are putting the number of sticks in a single row
        # like a list -> [1,3,5,7,....]
        self._k = k

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

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


    def play(self, ply: Nimply) -> None:
        row, num_objects = ply
        assert self._rows[row] >= num_objects
        assert self._k is None or num_objects <= self._k
        self._rows[row] -= num_objects

    def possible_plays (self) -> list:
        possiblePlays=[]
        th = 0
        if self._k != None:
            th = self._k
        else:
            th = max(self._rows)

        possiblePlays.append([Nimply(r,p+1) for r in range(self.num_rows) for p in range(self._rows[r]) if p+1 <= th or not self._rows ])
        return possiblePlays[0]
```

Task 3.1 Expert System

```python
def pure_random(state: Nim) -> Nimply:
```

```
    row = random.choice([r for r, c in enumerate(state._rows) if c > 0])
    num_objects = random.randint(1, state._rows[row])# now we are selecting a random number of sticks from the selected
row
    return Nimply(row, num_objects)
Game = Nim(4)
def calculate_nim_sum(rows):
    return reduce(lambda x, y: x ^ y, rows)


def expertSystem (Game: Nim) -> Nimply:
    best_ply = list()
    for i in Game.possible_plays():
        tmp = deepcopy(Game)
        tmp.play(i)
        best_ply.append((i,calculate_nim_sum(tmp._rows)))
    best_ply = sorted(best_ply,key= lambda x :x[1],reverse=False)
    retply=random.choice([num[0] for num in best_ply if num[1] == 0]) if best_ply[0][1]==0 else random.choice(best_ply)[0]
    return retply


strategy=[expertSystem,pure_random]

player = 0
while Game:
    ply = strategy[player](Game)
    # print(Game)
    Game.play(ply)
    print(f"status: After player {player} -> {Game}")
    player = 1 - player
winner = 1 - player
print(f"status: Player {winner} won!")
# print(ply)
## Task 3.2 : Evolved Rules
### Base-Nim Strategy
def decimal_to_base3(decimal_number):
    if decimal_number == 0:
        return '0'

    base3_digits = []
    while decimal_number > 0:
        decimal_number, remainder = divmod(decimal_number, 3)
        base3_digits.append(str(remainder))

    base3_number = ''.join(base3_digits[::-1])
    return base3_number

def convert_to_base_nim(rows):
    base_nim_sizes = [int(decimal_to_base3(num)) for num in rows]
    xor_sum = 0
    for number in base_nim_sizes:
        xor_sum ^= number
    return xor_sum


def Base_Nim(Game: Nim) -> Nimply:
    best_ply = list()

    for i in Game.possible_plays():
```

```
        tmp = deepcopy(Game)
        tmp.play(i)
        best_ply.append((i,convert_to_base_nim(tmp._rows)))
    best_ply = sorted(best_ply,key= lambda x :x[1],reverse=False)
    retply=random.choice([num[0] for num in best_ply if num[1] != 0]) if best_ply[0][1]!=0 else random.choice(best_ply)[0]
    return retply
Game = Nim(4)

strategy=[pure_random,Base_Nim]

player = 0
while Game:
    ply = strategy[player](Game)
    Game.play(ply)
    print(f"status: After player {player} -> {Game}")
    player = 1 - player
winner = 1 - player
print(f"status: Player {winner} won!")
## Task 3.3: minmax
def eval_terminal(Game):
    l = copy(Game._rows)
    o = reduce(lambda x, y: x ^ y, l)
    return 0 if not o else sum(Game._rows)
    # return sum(Game._rows)

game = Nim(4)
eval_terminal(game)
# o =reduce(lambda x, y: x ^ y, game._rows)
# # print(game._rows)
# print(sum(game._rows))
# print(o)
# 0 if not o else sum(game._rows)
def minmax(Game : Nim) -> Nimply:

    val = eval_terminal(Game)
    possible =  Game.possible_plays()
    if (val == 0 and sum(Game._rows) == 0) or len(possible) == 0 :
        return None,val

    evaluations = list()
    for ply in Game.possible_plays():
        if ply[0] != None:
            tmp = deepcopy(Game)
            tmp.play(ply)
            _,val = minmax(tmp)
            evaluations.append((ply, -val))

    s = random.choice([num[0] for num in evaluations if num[1] == 0 and num[0]!= None]) if evaluations[0][1]== 0 else list()
    return s if len(s)!=0 else max(evaluations,key= lambda k:k[1])[0]

Game = Nim(3)
strategy=[expertSystem,minmax]

player = 0
while Game:
    ply = strategy[player](Game)
    # print(ply)
    Game.play(ply)
```

```
    print(f"status: After player {player} -> {Game}")
    player = 1 - player
winner = 1 - player
print(f"status: Player {winner} won!")

## Task 3.4 : Reinforcement Learning
class RL:
    def __init__(self, Game, alpha=0.15, random_factor=0.2):  # 80% explore, 20% exploit
        self.Game = Game
        self.state_history = [(tuple(Game._rows), 0)]
        self.alpha = alpha
        self.random_factor = random_factor
        self.G = {}
        self.G [tuple(Game._rows)]=np.random.uniform(low =0.1,high=1.0)

    def choose_action(self):
        maxG = -10e15 # very low number

        allowedMoves = self.Game.possible_plays()
        next_ply = allowedMoves[0]
        randomN = np.random.random()
        if randomN < self.random_factor:
          row = random.choice([r for r, c in enumerate(allowedMoves)])
          next_ply = allowedMoves[row]
        else:
            for ply in allowedMoves:
                tmp = deepcopy(self.Game)
                tmp.play(ply)
                if self.G.get(tuple(tmp._rows)):
                    if self.G[tuple(tmp._rows)] >= maxG:
                        next_ply = ply
                        maxG = self.G[tuple(tmp._rows)]
                else:
                    self.G[tuple(tmp._rows)] = np.random.uniform(low =0.1,high=1.0)
        return next_ply
    def update_state_history(self, rows, reward):
        self.state_history.append((tuple(rows), reward))

    def learn(self):
        target = 0
        for prev, reward in reversed(self.state_history):
            if self.G.get(prev):
                self.G[prev] = self.G[prev] + self.alpha * (target - self.G[prev])
            else:
                self.G[prev] = np.random.uniform(low =0.1,high=1.0)

            target += reward # and here we are updating the reward as the cumulative reward

        self.state_history = []

        self.random_factor -= 10e-5  # decrease random factor each episode of play


Game = Nim(5)
rl = RL(Game)
count = 0

for i in range(5000):
```

```
   print ('**************New Game****************')
   Game = Nim(5)
   rl.Game = Game
   while Game:
       action = rl.choose_action()
       Game.play(action)
       reward = -1 if Game else 0
       state = Game._rows
       rl.update_state_history(state,reward)
       print(f"status: After RL  {Game}")
       if not Game:
           count += 1
           print('RL won')
           break
       ply = expertSystem(Game)
       Game.play(ply)
       print(f"status: After expert  {Game}")
       if not Game:
           print('Expert Won')

   rl.learn()
print( f'The number of times that RL won is {count} out of 5000 times')
```

```
    # 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 as np
import sys

new_limit = 50000  # Example: Setting a new recursion limit of 5000
sys.setrecursionlimit(new_limit)

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

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

  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)

class ExpertPlayer(quarto.Player):
  # here my algorithm extends the Player class and takes as an input the quarto object of Quarto class
  """Random player"""

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

  def is_a_win (self,l):
    high =0
    colored =0
    solid =0
    square =0
    for p_n in l:
      if p_n == -1:
        return False
      piece =self.get_game().get_piece_charachteristics(p_n)
      high += 1 if piece.HIGH else 0
      colored += 1 if piece.COLOURED else 0
      solid += 1 if piece.SOLID else 0
      square += 1 if piece.SQUARE else 0
    if high ==4 or colored ==4 or solid ==4 or square ==4:
      return True
    else:
      return False

  def get_score(self,l) -> int:
    high =0
    colored =0
    solid =0
    square =0
    score =0
```

```
        for p_n in l:
            if p_n != -1:
                piece =self.get_game().get_piece_charachteristics(p_n)
                high += 1 if piece.HIGH else 0
                colored += 1 if piece.COLOURED else 0
                solid += 1 if piece.SOLID else 0
                square += 1 if piece.SQUARE else 0

        score +=1 if high ==3 else 0
        score +=1 if colored ==3 else 0
        score +=1 if solid ==3 else 0
        score +=1 if square ==3 else 0
        return score

    def piece_score (self,p) -> int:
        game = self.get_game()
        s =0
        if p in game._board:
            # this piece is not available
            return -1
        # if the piece will make a win i return -2
        for i,row in enumerate(game._board):
            if -1 in row:
                for j,e in enumerate(row):
                    if e == -1 :
                        game._board[i,j] = p
                        reversed_arr = np.fliplr(game._board)
                        if self.is_a_win(row) or self.is_a_win(game._board[:,j]) or self.is_a_win(game._board.diagonal()) or
self.is_a_win(reversed_arr.diagonal()):
                            game._board[i,j] = -1
                            return -2
                        else:
                            s += self.get_score(row) +self.get_score(game._board[:,j]) +self.get_score(game._board.diagonal()) +
self.get_score(reversed_arr.diagonal())
                        game._board[i,j] = -1
        return s

    def choose_piece(self) -> int:
        # i will choose the piece that max the reward without making the win
        # loop for all pieces
        game = self.get_game()
        available_pieces = list()

        for j in range(16):
            if j not in game._board:
                score= self.piece_score(j)
                available_pieces.append((j,score))

        return sorted(available_pieces,key= lambda x:x[1],reverse=True)[0][0]

    def place_score(self,place,piece)-> int:
        i,j = place
        game = self.get_game()
        game._board[i,j] = piece
        s=0

        for row in game._board:
```

```
        reversed_arr = np.fliplr(game._board)
        if self.is_a_win(row)  or self.is_a_win(game._board.diagonal()) or self.is_a_win(reversed_arr.diagonal()):
            return -2
        else:
            s += self.get_score(row) +self.get_score(game._board.diagonal()) + self.get_score(reversed_arr.diagonal())
    for j in range(4):
        if self.is_a_win(game._board[:,j]):
            return -2
        else:
            s+= self.get_score(game._board[:,j])
    return s

  def place_piece(self) -> tuple[int, int]:
    # check for the places that guarantees the win if not
    # place the piece in the place that after that will guarantee a low value
    game = self.get_game()
    piece = game.get_selected_piece()
    if piece == -1:
        return random.randint(0, 3), random.randint(0, 3)
    available_places = list()

    for i in range(4):
        for j in range(4):
            if game._board[i,j] == -1 :
                p = i,j
                score= self.place_score(p,piece)
                game._board[i,j] = -1
                if(score == -2):
                    return int(p[1]),int(p[0])
                available_places.append((p,score))
    place = sorted(available_places,key=lambda x:x[1])[0][0]
    return int(place[1]) , int(place[0])

  def get_game(self):

    return super().get_game()

class MinMaxPlayer(quarto.Player):
  """Min Max approach player"""

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

  def is_a_win (self,l):
    high =0
    colored =0
    solid =0
    square =0
    for p_n in l:
        if p_n == -1:
            return False
        piece =self.get_game().get_piece_charachteristics(p_n)
        high += 1 if piece.HIGH else 0
        colored += 1 if piece.COLOURED else 0
        solid += 1 if piece.SOLID else 0
        square += 1 if piece.SQUARE else 0
    if high ==4 or colored ==4 or solid ==4 or square ==4:
        return True
```

```
    else:
        return False

def get_score(self,l) -> int:
    high =0
    colored =0
    solid =0
    square =0
    score =0

    for p_n in l:
        if p_n != -1:
            piece =self.get_game().get_piece_charachteristics(p_n)
            high += 1 if piece.HIGH else 0
            colored += 1 if piece.COLOURED else 0
            solid += 1 if piece.SOLID else 0
            square += 1 if piece.SQUARE else 0

    score +=1 if high ==3 else 0
    score +=1 if colored ==3 else 0
    score +=1 if solid ==3 else 0
    score +=1 if square ==3 else 0

    return score

def current_game_score(self)->int:
    game = self.get_game()
    s=0
    for row in game._board:
        if -1 not in row :
            if self.is_a_win(row):
                return -2
        s += self.get_score(row)
    for i in range(4):
        if -1 not in game._board[:,i] :
            if self.is_a_win(game._board[:,i]):
                return -2
        s+= self.get_score(game._board[:,i])
    if self.is_a_win(game._board.diagonal()):
        return -2
    else:
        s += self.get_score(game._board.diagonal())

    reversed_arr = np.fliplr(game._board)
    if self.is_a_win(reversed_arr.diagonal()):
        return -2
    else:
        s += self.get_score(reversed_arr.diagonal())
    return s

def piece_score (self,p) -> int:
    game = self.get_game()
    s =0
    if p in game._board:
        return -1
    # if the piece will make a win i return -2
    for i,row in enumerate(game._board):
        if -1 in row:
```

```
            for j,e in enumerate(row):
                if e == -1 :
                    game._board[i,j] = p
                    reversed_arr = np.fliplr(game._board)
                    if self.is_a_win(row) or self.is_a_win(game._board[:,j]) or self.is_a_win(game._board.diagonal()) or
self.is_a_win(reversed_arr.diagonal()):
                        game._board[i,j] = -1
                        return -2
                    else:
                        s += self.get_score(row) + self.get_score(game._board[:,j]) +self.get_score(game._board.diagonal()) +
self.get_score(reversed_arr.diagonal())
                    game._board[i,j] = -1
        return s

    def MinMax_piece(self):
        game = self.get_game()
        available_pieces = list()
        for j in range(16):
            if j not in game._board:
                available_pieces.append(j)
        old_piece = game.get_selected_piece()
        eval= self.current_game_score()
        if eval == -2 or len(available_pieces)==0:
            game.select(old_piece)
            return None,eval

        evaluations = list()
        for piece in available_pieces:
            o_p = game.get_selected_piece()
            game.select(piece)
            x,y =self.place_extreme_piece()
            game._board[y,x] = piece
            _,val = self.MinMax_piece()
            if np.count_nonzero(game._board == -1) == 1:
                evaluations.append((piece,-2))
            else:
                val= self.current_game_score()
                evaluations.append((piece,val))
            game._board[y,x]= -1
            game.select(o_p)
        game.select(old_piece)
        selected_piece = sorted(evaluations,key= lambda x:x[1], reverse=True)[0]
        return selected_piece

    def choose_extreme_piece(self,t=True)-> int:
        # t=True he will select the piece with highest score 8,7,-1  then 8
        # here i want to choose a piece that will gaurantee for him the loose
        game = self.get_game()
        available_pieces = list()
        for j in range(16):
            if j not in game._board:
                score= self.piece_score(j)
                available_pieces.append((j,score))
        available_pieces =sorted(available_pieces,key= lambda x:x[1],reverse=t)
        return available_pieces[0][0]

    def choose_piece(self) -> int:
        # my goal here is to choose the piece that gives
```

```
        # the min reward
        num = self.MinMax_piece()
        return num[0]

    def place_score(self,place,piece)-> int:
        i,j = place
        game = self.get_game()
        game._board[i,j] = piece
        s=0

        for row in game._board:
            reversed_arr = np.fliplr(game._board)
            if self.is_a_win(row)  or self.is_a_win(game._board.diagonal()) or self.is_a_win(reversed_arr.diagonal()):
                # game.print()
                return -2
            else:
                s += self.get_score(row) +self.get_score(game._board.diagonal()) + self.get_score(reversed_arr.diagonal())
        for j in range(4):
            if self.is_a_win(game._board[:,j]):
                return -2
            else:
                s+= self.get_score(game._board[:,j])
        # no winning place

        return s

    def place_extreme_piece(self) -> tuple[int, int]:
        # he will place it in the best place
        game = self.get_game()
        piece = game.get_selected_piece()
        if piece == -1:
            return random.randint(0, 3), random.randint(0, 3)
        available_places = list()
        for i in range(4):
            for j in range(4):
                if game._board[i,j] == -1 :
                    p = i,j
                    score= self.place_score(p,piece)
                    game._board[i,j] = -1
                    if score == -2:
                        return int(p[1]),int(p[0])
                    if score != -1:
                        available_places.append((p,score))
        place = sorted(available_places,key=lambda x:x[1],reverse=False)[0][0]
        # print(place)
        return int(place[0]) , int(place[1])

    def MinMax_place(self):
        game = self.get_game()
        piece = game.get_selected_piece()
        if piece == -1:
            return random.randint(0, 3), random.randint(0, 3)
        available_places = list()
        for i in range(4):
            for j in range(4):
                if game._board[i,j] == -1 :
                    p = i,j
                    available_places.append(p)
```

```
        val = self.current_game_score()

        if val == -2 or len(available_places) == 1 :
            return available_places[0], val

        evaluations= list()
        for place in available_places:
            game._board[place[0],place[1]] = piece
            o_p = game.get_selected_piece()
            e = self.choose_extreme_piece()
            game.select(e)
            _,val = self.MinMax_place()
            if np.count_nonzero(game._board == -1) == 1:
                evaluations.append((place,-2))
            else:
                val= self.current_game_score()
                evaluations.append((place,val))
            game._board[place[0],place[1]] = -1
            game.select(o_p)
        game.select(piece)

        selected_place=0
        if isinstance(evaluations, list):
            selected_place = min(evaluations,key= lambda x:x[1])
        else:
            selected_place = evaluations
        return selected_place

    def place_piece(self) -> tuple[int, int]:
        game = self.get_game()
        if np.count_nonzero(game._board == -1)==1:
            for i in range(4):
                for j in range(4):
                    if game._board[i,j] ==-1:
                        return j,i

        p = self.MinMax_place()
        r=None
        if isinstance(p, tuple):
            r=p[0]
        else:
            r=p
        return r[1],r[0]

    def get_game(self):

        return super().get_game()

def main():
    game = quarto.Quarto()

    # # Making game easier for MinMax Algorithm to work
    # # " Uncomment the next section and comment the ExpertPlayer line to test the MinMax "
    # game.select(10)
    # game.place(1,0)
    # game.select(7)
    # game.place(2,0)
    # game.select(9)
```

```
    # game.place(0,1)
    # game.select(3)
    # game.place(1,1)
    # game.select(11)
    # game.place(3,1)
    # game.select(8)
    # game.place(0,2)
    # game.select(4)
    # game.place(1,2)
    # game.select(12)
    # game.place(1,3)
    # game.select(0)
    # game.place(3,3)
    # game.set_players(( RandomPlayer(game),MinMaxPlayer(game)))

    game.set_players((ExpertPlayer(game), RandomPlayer(game)))
    winner = game.run()
    logging.warning(f"main: Winner: player {winner}")

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()
```