report.md

Report - Vincenzo Micciche'

Set Covering

In the first lab we had to develop a strategy to cover a set with as few tiles (with different shapes) as possible I tried 4 different approaches for solving the task.

- At first a Greedy Solution calculated as a subtraction from the value of the state of the distance from the state (trying to maximize the number of tiles possible)
- An informed one with the distance from the goal
- A* with an heuristic of "How many of the largest tiles would I need to reach the goal?"
- Something that was called wrong A* but with a non-admissible heuristic, and more similar to the greedy, it reached a solution in less steps but it was not the optimal, it favours the best state with less tiles

Data Structures and Utils functions

The frontier was a priority Queue with 3 values, one was the priority and the value depends on the algorithm, one is the state of the taken and not taken elements and a memory value used in some of the algorithms.

```
    frontier.put((0,[set(), set(range(NUM_SETS))] , 0))

    t=frontier.get()
    current_state=t[1]
    print(state_to_str(current_state))
```

A function to get the largest tile (was used in the A*):

```
    covered=get_covered(state)

    d=0
    for _,i in enumerate(state[1]):
        temp=np.count_nonzero( np.logical_and(SETS[i], np.logical_not(covered)) ==True)
        if(temp>d):
            d=temp
    return d
```

A function to get how many elements are left to cover

```
def get_distance_from_goal(state):
    t=get_covered(state)
    return np.count_nonzero(t==False)
```

The Loop

At each iteration of the algorithm adds to the frontier every reachable state from the current state with the calculated priority for that state according to the algorithm.

At the end chooses the best one (the one with lowest priority value) as current state and checks if the goal has been reached.

Greedy

```
# --- Greedy Solution --- #
    d_to_next_state=t[2]- get_distance_from_state(current_state,action)
    frontier.put((d_to_next_state,new_state, d_to_next_state))
```

The priority in the frontier depends on how many more elements does the state cover with respect to the previous state. (The third value is unused)

Informed

```
# --- Informed Solution --- #
    frontier.put((get_distance_from_goal(new_state), new_state, t[2]))
```

Uses the information of how many elements are left to cover as a priority. (The third value is unused)

A*

```
if largest_tile!=0:
    v=get_distance_from_goal(new_state)//largest_tile + len(new_state[0])
    frontier.put((v, new_state , v))
```

Makes sure that the largest tile is not equal to zero and estimates how many tiles with the size of the largest would be required to reach the goal. To that the length of the taken set is added to give higher priority to states with less taken elements.

Another Greedy attempt

```
d_to_next_state=t[2]- get_distance_from_state(current_state,action)
if(d_to_next_state!=0):
    frontier.put((  d_to_next_state+len(new_state[0]), new_state, d_to_next_state))
```

It's again similar to the greedy function (that I mistakenly called wrong A* in the code) but should prefer states with less taken tiles. The result is usually the same to the Greedy

Set Covering With Tweaking

Takes and leaves random tiles until it covers the whole set, then it starts improving the solution by taking cover sets with less tiles for a total of 100 tweaks in my solution.

The tweak function moves one tile from the non-taken set to the taken set or vice-versa, evaluates the number of taken elements and returns a value. If elements are still uncovered then the value will be -1000 (to make sure it was small enough with respect to states that fully cover all the elements). Otherwise the negative length of the state will be the value returned.

```
def tweak(state):
    new_state=copy(state)
    action=int(random()*NUM_SETS)
    new_state[0]^={action}
    new_state[1]^={action}

    if count_false(new_state)>0:
        return new_state, -1000
    else :
        return new_state,-len(new_state[0])
```

In the loop the tweak funcition will be called at each iteration, but will override the previous one only if the value is larger than the previous (closer to 0), thus will look at the state with less taken elements.

Lab2 - Nim

I tried to solve the lab without looking up on the solution for the algorithm, and I found that my algorithm was working well (even though it was not the best). I tried different evolution strategies to maximize the probability to choose one of the algorithm pre-written, neither of them was working or converging to a solution and I found out that the problem was an oversight on the tweak function and the softmax making the probabilities covnerge to equal numbers. An attempt was made at creating a genetic algorithm, but the result was not as good as expected

My algorithm

```
def vinzgorithm(state: Nim) -> Nimply:
    ns=nim_sum(state)

    if(get_number_of_rows(state)>=2):
        return Nimply(int(np.argmax(state.rows)), int(np.max(state.rows)))
    else:
        moves=[Nimply(r,c-1) for r,c in enumerate(state.rows) if(c>1)]

    logging.info(f"len: {len(moves)} nin_sum: {ns}")
    if(len(moves)==0):
        moves=[Nimply(n,o) for n,c in enumerate(state.rows) for o in range(1,c+1)]
    ply=random.choice(moves)
    return ply
```

If there is more than one row, takes the whole row. If there is only one row leaves only one stick in the row. If there is only one stick it already lost.

The Individual

```
class es_individual:
    def __init__(self, *strategies, compare_strategy=optimal):
        self.strategies=strategies
        self.vec=np.array([random.random() for _ in range(len(strategies))])
        self.vec=softmax(self.vec)
        self.fitness_value=0
        self.fitness(compare_strategy)

    def strategy(self, state:Nim):
        s=np.random.choice(self.strategies, p=self.vec, replace=False)
        assert callable(s), f"strategy not callable, type: {type(s)}"
        return s(state)

    def fitness(self, strategy=optimal):
        _,win1=play(strategy, self.strategy, times=100)
        if(win1>self.fitness_value):
            fitness_value=win1
            return True
        return False

    def tweak(self, strategy=optimal, sigma=1):
        new_ind=deepcopy(self)
        for i in range(len(self.vec)):
            new_ind.vec[i]+=random.gauss(0,sigma)
        new_ind.vec=softmax(new_ind.vec)
        new_ind.fitness(strategy)
        return new_ind
```

The individual object will take a set of strategies to choose among and will choose a random one with weighted probabilities. The weigthed probabilities are learnt with evolution strategies.

```
def evolve_first_improv(epochs, nim_strategy=optimal) -> es_individual:
    individual=es_individual(pure_random, vinzgorithm, optimal, gabriele)
    sigma=1
    for i in range(epochs):
        sigma=(epochs-i)/epochs
        new_individual=individual.tweak(optimal,sigma)
        if new_individual.fitness_value>individual.fitness_value:
            individual=new_individual
    return individual

def evolve_steepest(epochs,samples, nim_strategy=optimal) -> es_individual:
    individual=es_individual(pure_random, vinzgorithm, optimal, gabriele)
    for i in range(epochs):
        sigma=(epochs-i)/epochs
        new_individual=individual.tweak(nim_strategy, sigma)

        for _ in range(samples-1):
            temp_individual=new_individual.tweak(nim_strategy,sigma)
            if temp_individual.fitness_value>new_individual.fitness_value:
                new_individual=temp_individual

        if new_individual.fitness_value>individual.fitness_value:
            individual=new_individual

    return individual

def evolve_comma_lambda(epochs, samples, nim_strategy=optimal) -> es_individual:
    individual=es_individual(pure_random, vinzgorithm, optimal, gabriele)
    for i in range(epochs):
        sigma=(epochs-i)/epochs
        new_individual=individual.tweak(nim_strategy,sigma)

        for _ in range(samples-1):
            temp_individual=new_individual.tweak(nim_strategy, sigma)
            if temp_individual.fitness_value>new_individual.fitness_value:
                new_individual=temp_individual
```

```
        individual=new_individual

    return individual
```

These are the functions I used to evolve the strategy.

I realized too late that in the committed file I had forgotten "self." before fitness value and for to reasons before mentioned it didn't work and the softmax converges towards equal probabilities. With those error fixed, as Alex Buffa mentioned in his issue the algorithm converges towards vinzgorithm.

Genetic Attempt

```
SIZE=3

default_state=Nim(SIZE)

possible_states=enum_state(default_state.rows)

class Individual:
    genome=dict()
    def __init__(self):
        global possible_states, default_state

        for t in possible_states:
            moves=[Nimply(r,o) for r,c in enumerate(t) for o in range(1,c+1)]
            if(len(moves)!=0):
                self.genome[t]=random.choice(moves)
            else:
                self.genome[t]=Nimply(0,0)

    def strategy(self, state:Nim):
        return self.genome[state.rows]

def crossover(individual1:Individual, individual2:Individual):
    global possible_states
    individual3=Individual()
    individual4=Individual()
    for t in possible_states:
        if random.choice([True,False]):
            individual3.genome[t]=individual1.genome[t]
            individual4.genome[t]=individual2.genome[t]
        else:
            individual4.genome[t]=individual1.genome[t]
            individual3.genome[t]=individual2.genome[t]
    return individual3, individual4

def compare(individual1, individual2):
    nim=deepcopy(default_state)

    strategy=(individual1,individual2)
    assert type(individual1)==Individual
    assert type(individual2)==Individual
    assert type(strategy[0])==Individual
    player = 0
    while nim:
        ply = strategy[player].genome[nim.rows]
        nim.nimming(ply)
        player = 1 - player
    return strategy[player]


def eval(individual1, test):
    nim=deepcopy(default_state)

    strategy=(test, individual1.strategy)
    player = 0
    while nim:
        ply = strategy[player](nim)
        nim.nimming(ply)
        player = 1 - player
```

```
    return player==1

def evolve(power_of_two:int):
    n=2**power_of_two
    individuals=[Individual() for _ in range(n)]

    while(True):
        new_gen=[]
        for i in range(0,n,2):
            new_gen.append(compare(individuals[i], individuals[i+1]))

        if(len(new_gen)==1):
            return new_gen[0]

        individuals=[]
        for j in range(0,len(new_gen),2):
            individuals.extend(crossover(new_gen[j], new_gen[j+1]))

        n=len(individuals)


def evolve2(power_of_two:int):
    n=2**power_of_two
    individuals=[Individual() for _ in range(n)]

    while(True):
        new_gen=[]
        for i in range(0,n,2):
            if(eval(individuals[i], optimal)):
                new_gen.append(individuals[i])

        if(len(new_gen)==1):
            return new_gen[0]
        if(len(new_gen)==0):
            return individuals[0]

        individuals=[]
        for j in range(0,len(new_gen),2):
            if j+1==len(new_gen):
                individuals.append(new_gen[j])
            else:
                individuals.extend(crossover(new_gen[j], new_gen[j+1]))

        n=len(individuals)
```

It was a first attempt at Genetic Algorithms, but I lacked the knowledge to improve it. The genome was defined as a combination move-state, I only did the crossover at the time of the commit and then tried the mutation. It still wasn't enough and it required to much time to evolve (I failed miserably).

For this lab I reviewed only Paolo Magliano: https://github.com/paolo-magliano/Computational-Intelligence/issues/1

```
Regarding the Lab02 - Nim, this is my review:

Event though the adaptive strategy doesn't have actual real parameters, the implementation of the genetic
First thing I noticed is the idea of how to represent the genome in a multidimensional array getting away
Then I noticed the choice of mutating the losing strategies instead of throwing them away, which balances
Maybe the "record" could have been used to crossover only relevant genes that were used for the phenotype
The code is well written but it could use a bit more comments to make it more readable.

Best Regards,
Vincenzo Micciche'
```

Lab 3/9

I started with the basics, just mutation and got not very good results. Then added crossover but still mutation seemd to give better results. But combining them with islands and mass extinction let the algorithm converge to solution for problem 2 and 5 with respectively 1.3M fitness calls and 1.5M calls. Never reached a solution for problem 10.

```
class Mutation():
    def __init__(self, start_population: int, kill_perc: float, mut_prob: float, gene_mut_prob:float, fi
        self.population=gen_population(start_population, fitness)
        self.mut_prob=mut_prob
        self.gene_mut_prob=gene_mut_prob
        self.kill_perc=kill_perc
        self.epochs=0
        self.verbose=verbose
        self.fitness=fitness
        self.with_removal=with_removal
        self.max_population=max_population

        print_population(self.population, self.verbose)

    def step(self):
        global mutate

        self.epochs+=1

        ### KILL ###
        kill_population(self.population, self.kill_perc, self.max_population)

        ### MUTATE ###
        mutate_population(self.population, self.mut_prob, self.gene_mut_prob, self.fitness, self.with_re
                    #print(f"After: {i[0]:.2%} {print_ind(i[1])}")
        print("Epoch: ", self.epochs," individuals: ", len(self.population), " best fitness: ", self.bes
        #print(print_ind(self.best[1]))
        print_population(self.population, self.verbose)

    @property
    def best(self):
        return self.population[0]

def kill_population(population:list, kill_perc:float, max_pop=None):
    to_remove=int(len(population)*kill_perc)

    for _ in range(to_remove):
        population.pop()
    if max_pop!=None:
        while(len(population)>max_pop):
            population.pop()



def mutate_population(population,  population_sub:list, mut_prob: float, gene_mut_prob:float, fitness: ca
    global gene_prob
    mutations=0
    for i in population_sub:
        #prob=(1 - (i[0]/100)) * mut_prob
        prob_ind=[0 for _ in range(1000)]
        prob=mut_prob
        if(random()<prob):
            t=mutate(i[1], gene_mut_prob*(1-i[0]))
            t_fit=fitness(t)
            if t_fit>i[0]:
                mask=mutation_mask(i[1], t)
                gene_prob=reapply_mutation(mask, gene_prob)
            if(with_removal):
                population.remove(i)
            population.add([t_fit,t.tolist()])
            mutations+=1
```

The first attempt was with just mutation to test how good it was, at each step a percentege of the population is killed and the remaining elements are mutated according to the fitness value. The higher it is, the less probable it is that a gene mutates.

```
class Crossover():
    def __init__(self, start_population: int, kill_perc: float, offspring: int, mut_prob: float, gene_mut
        self.population=gen_population(start_population, fitness)
        self.kill_perc=kill_perc
        self.offspring=offspring
        self.mut_prob=mut_prob
```

```
        self.gene_mut_prob=gene_mut_prob
        self.fitness=fitness
        self.with_removal=with_removal
        self.verbose=verbose
        self.max_population=max_population
        self.epochs=0

        print_population(self.population, self.verbose)

    def step(self, elite_threshold=0.2, losers_pool=None, max_loser=600):
        global mutate, crossover, loser_population

        self.epochs+=1
        elite=int(elite_threshold*len(self.population))
        n_couples=int(0.5*elite*0.5)

        for _ in range(n_couples):
            crossover_population_by_difference(self.population, self.population, self.offspring, self.fi

        mutate_population(self.population, self.population, self.mut_prob, self.gene_mut_prob,self.fitnes

        if losers_pool is None:
            kill_population(self.population, self.kill_perc,self.max_population)
        else:
            loser_population(self.population, losers_pool, self.kill_perc,self.max_population, max_loser

        print("Epoch: ", self.epochs," individuals: ", len(self.population), " best fitness: ", self.best
        #print(print_ind(self.best[1]))
        print_population(self.population, self.verbose)
    @property
    def best(self):
        return self.population[0]


def inject_new_individuals(population:SortedList, fitness, n=5):
    for _ in range(n):
        ind=choices([0, 1], k=1000)
        fit=fitness(ind)
        population.add([fit,ind])

def crossover_population(population_sub, population,offspring, fitness):
    #selected=[0 for _ in population]
    #print("Selected len: ", len(selected), " population: ", len(population))

    p1=0
    p2=0
    #while(True):
    p1=randint(0,len(population_sub)-1)
    p2=randint(0,len(population_sub)-1)
        # print("P1: ",p1," P2: ",p2, len(population))
        # if p2!=p1 and (selected[p1]==0) and (selected[p2]==0):
        #     break
    #selected[p1]=1
    #selected[p2]=1

    parent1=population_sub[p1]
    parent2=population_sub[p2]

    for _ in range(offspring):
        ind=crossover(parent1[1],parent2[1]).tolist()
        fit=fitness(ind)
        if ind!= parent1[1] and ind != parent2[1]:
            population.add([fit,ind])

def crossover_population_by_difference(population_sub, population,offspring, fitness):
    #selected=[0 for _ in population]
    #print("Selected len: ", len(selected), " population: ", len(population))

    p1=0
    p2=0
    #while(True):
```

```
    p1=randint(0,len(population_sub)-1)
    p2=0
        # print("P1: ",p1," P2: ",p2, len(population))
        # if p2!=p1 and (selected[p1]==0) and (selected[p2]==0):
        #     break
    #selected[p1]=1
    #selected[p2]=1
    parent1=population_sub[p1]
    parent2=population_sub[p2]
    maxent=0
    for i in range(len(population_sub)):
        temp=population_sub[i]
        e=entropy(parent1[1],temp[1])

        if e>maxent:
            maxent=e
            parent2=temp

    #print(f"{print_ind(temp[1])}, {e}")

    for _ in range(offspring):
        ind=crossover(parent1[1],parent2[1]).tolist()
        if ind!= parent1[1] and ind != parent2[1]:
            fit=fitness(ind)
            population.add([fit,ind])

def entropy(ind1, ind2):
    return sum([1 for i in range(len(ind1)) if ind1[i]!=ind2[1]])
```

For the crossover I defined an entropy function that calculates how different 2 individuals are and chooses them to do a crossover, then mutates the population. In this class there is also a function for the injection of the individuals which will be used in the final loop.

```
cross1=Crossover(50, 0.5, 15, .8, .5, fitness, verbose=False, with_removal=False,max_population=200)
cross2=Crossover(50, 0.5, 15, .8, .5, fitness, verbose=False, with_removal=False,max_population=200)


for i in range(10000):
    cross1.step(.2)
    cross2.step(.2)

    remove_similar(cross1.population)
    remove_similar(cross2.population)

    if (i+1 )% 40==0:
        kill_population(cross1.population, 0.95,200)
        kill_population(cross2.population, 0.95, 200)
        inject_new_individuals(cross2.population,fitness, 30)
        inject_new_individuals(cross1.population,fitness, 30)
        cross1.population.update(cross2.population[:5])
        cross2.population.update(cross1.population[:5])

    print(f"\nCalls {fitness.calls}")
    if cross1.best[0]==1.0 or cross2.best[0]==1.0:
        break
```

In the final implementation 2 isalnds are defined (the "Crossover" objects), an epoch is called for both of them and the equal individuals are removed in order to keep a certain level of entropy in the populations and every 40 epochs most of the population is killed and new individuals are injected in both the isalnds and the top 5 elements in each island is copied and injected in the other. The loop is interrupted when one of the populations has an individual with 1.0 fitness.

Reviews for this Lab

To Mattia Sabato

```
I tried to understand as much as possible to learn from you, and it seems to me that you decided to trad
Clever implementation of the most useful strategies for evolving the population.
It took me a while since there's not much documentation, with that said I'm looking forward to read a ve
```

To Alex Buffa

```
The code is well written even though not commented, and a bit hard to comprehend, but overall a good wor
```

Lab 10

I wanted to test how well was the Q-learning method and for the tictactoe and it gave pretty good results.

Since I started early I wrote everything from scratch, I followed the advice to start with a magic box:

```
self.board=np.array([[1,6,5],[8,4,0],[3,2,7]])
```

And the state is made of two sets one for player 0 taken and one for player 1. The game ends when the sum of one of the sets is 12 or the board is full.

There are 2 agents (same class) playing against each other with different tables:

```
class Agent1():

    def mask_selections(lookup, mask:bool):
        if mask:
            return lookup
        else:
            return -math.inf

    mask_selection=np.vectorize(mask_selections)

    def __init__(self,player ,max_mem=10000, decay=DECAY):
        self.player=player
        self.moves_done=0
        self.memory=[]
        self.lookup=dict() #Lookup table for the quality state-value
        self.state_frequency=dict()
        self.eps=0
        self.was_random=False
        self.decay=decay
    # The VALUE function takes in a STATE and gives the discounted reward obtainable from that state fol
    # The QUALITY function takes in a STATE and an ACTION and gives the discounted reward obtainable fro

    def step(self, board:Environment):

        available_mask=[i==-1 for i in board.state.reshape([9])]
        if not np.any(available_mask):
            return True, None
        self.state=deepcopy(board.taken_sets)
        self.action, self.was_random = self.select_action(board, available_mask,0.0,0.9,self.decay)
        res=board.play(self.player, self.action)

        return False, self.was_random
        #print(available_mask)

    def update_values(self):
        reward, final=board.check_win_player(self.player)
        if TRAINING:
            self.memory.append((self.state, self.action, reward))

        if final:
            self.memorize()
            self.memory.clear()
            return True
        else:
            return False


    # Saves in a lookup table the cumulative reward MEAN over all the times
    # a certain action has been called on that state

    def memorize(self):  #GOT IT MEMORIZED???
        if not TRAINING:
            return
        cumulative_reward=0
        while len(self.memory)>0:
            taken_sets, action, reward=self.memory.pop()

            cumulative_reward=reward+DISCOUNT*cumulative_reward
```

```
            if taken_sets not in self.lookup.keys():
                self.lookup[taken_sets]=[0 for _ in range(9)]
                self.state_frequency[taken_sets]=[0 for _ in range(9)]

            assert self.lookup[taken_sets][action] is not math.nan, f"Not a Number error {self.state_fre


    def calc_value(self, taken_sets, available_mask) -> int:
        if(taken_sets in self.lookup.keys()):
            values=self.mask_selection(self.lookup[taken_sets], available_mask)
            return np.argmax(values), False
        else:
            return random.sample(range(9), k=1, counts=map(lambda s:1 if s else 0, available_mask))[0],


    def select_action(self, board:Environment, available_mask, eps_end, eps_start, eps_decay):

        if TRAINING:
            self.eps=eps_end+math.exp(-1*self.moves_done/eps_decay)*(eps_start-eps_end)
        else:
            self.eps=0
        self.moves_done+=1
        if(random.random()<self.eps):
            return random.sample(range(9), k=1, counts=map(lambda s:1 if s else 0, available_mask))[0],
        else:
            return self.calc_value(board.taken_sets, available_mask)
```

Every agent has a dictionary of states and value and the table is updated according to Bellman equation.

Training Phase

The two agents play against each other, at first `self.eps=eps_end+math.exp(-1*self.moves_done/eps_decay)*(eps_start-eps_end)` this function determines the decreasing probability of choosing a random action, and moreover if the dictionary does not contain an entry for the current state the move will be random. The states and actions are saved in a memory and when the game is over the memory is unrolled to update the values: `cumulative_reward=reward+DISCOUNT*cumulative_reward` with this function the discounted reward is calculated and added to the overall value for that action at that state.

Playing Phase

The agent chooses the action with highest value.

Review for this Lab

To Abolfazl Javidian

```
Hope the holidays went well,
The code is very organized and easy to understand, some comment would just make it faster to understand.
The results are very good and I think the soft update really helps a lot, maybe you could have try to ex
```

To Arild Strømsvåg

```
The code is very readable even with few comments, very modular and easy to debug, and by the results it
Maybe it could converge faster to a solution if the epsilon of the learning agent started at 1 and then
```

Snake

Did a presentation on Deep-Q-learning applied to snake, did a pull request relative to that.

My algorithm implementations uses a Network of 3 layers that is updated at every step and then every time the game ends.

```
class SimpleAgent(nn.Module):
    def __init__(self):
        super().__init__()
        self.flat=nn.Flatten()
        self.layer1=nn.Linear(14,256) # 5x5
        self.layer2=nn.Linear(256,128)
        self.final=nn.Linear(128,4)

    def forward(self, in_data) -> torch.Tensor:
        x=self.flat(in_data)
```

```
        x=self.layer1(x)
        x=F.relu(x)
        x=F.relu(self.layer2(x))
        x=self.final(x)
        return x
```

The memory is unrolled at every game end to update the network using the Bellman Equation.

It uses Epsilon-Greedy Exploration Strategy (with linear decrease) to train the network on "wrong" actions.

To stabilize the training two twin network are used, one, the main will be fitted on the temporal difference calculated on the second, the target, with the following formula:

$$R_t + \gamma \max_a Q(s_{t+1}, a)$$

Rt + γ amax Q(st+1, a)

The snake is rewarded more when it eats an apple, and is penalized when it hits a wall or itself or if it survives too long with the same length (it basically starves) to avoid it circling around infinitely.

It's important how to code the state, in this case the state is composed of the 8 neighbouring pixels, the one hot encoding of the direction and the relative distance from the apple.

Quixo

For the quixo I implemented the minimax, the alpha beta pruning, the Montecarlo tree seach and 2 types of reinforcement learning: Advantage Actor Critic and Deep Q Leaning. Minimax and Alpha-Beta are very slow even with a depth of 2-3 maybe I wasn't very good at optimizing it Montecarlo was acceptable A2C worked only once in a while when the training was going good rapidly Deep Q Learning works pretty well and the resulting agent is incredibly fast (and with a winrate of 90-95%)

Loss

The loss for minimax, montecarlo and alpha-beta pruning was simply the max count of symbols in line and uninterrupted.

Minimax

```
class MiniPlayer(Player): #minimax with alpha-beta (TODO)
    def __init__(self, player_id=0):
        self.player_id=player_id
        pass

    def make_move(self, game: Game, alphabeta=False) -> tuple[tuple[int, int], Move]:
        global moves

        values=np.zeros(len(moves))
        for i, m in enumerate(moves):
            temp_g=deepcopy(game)
            ok=temp_g._Game__move(m.position, m.move,self.symbol)
            temp_g.current_player_idx=1-temp_g.current_player_idx
            if ok:
                values[i]=self.__minimax_node(temp_g, 1, 1-self.symbol)


        idx=np.argmax(values)

        #print("returned")
        return moves[idx]



    def __minimax_node(self, game:Game,n=4, turn=1) -> int:
        global moves
        if n<0:
            return loss_v1(game)

        outs=[]
        for m in moves:
            temp_g=deepcopy(game)

            assert turn==temp_g.get_current_player(),"Wrong turn"
            ok=temp_g._Game__move(m.position, m.move,turn)
```

```
            temp_g.current_player_idx=1-temp_g.current_player_idx
            if ok:
                t=self.__minimax_node(temp_g, n-1, 1-turn)
                outs.append(t)
                #print(outs)

        if len(outs)==0:
            print(f"Hit bottom at {n}")
            return -1

        if turn==1-self.symbol:
            return min(outs)
        elif turn==self.symbol:
            return max(outs)
```

Here the tree is developed as a recursive function __minimax_node where, if it's a leaf (meaning it reached max depth), returns the loss value of the state otherwise calculates the max of the children if it's agent's turn or the minimum if it's other player's turn. The drawback is an high memory consumption and high time to calculate.

Alpha Beta Pruning

```
class AlphaPlayer(Player):
def __init__(self, player_id, depth=2):
    self.player_id=player_id
    self.depth=depth

def make_move(self, game: Game) -> tuple[tuple[int, int], Move]:
    print("Move")
    t=None
    outs=dict()

    for m in moves:
        temp_g=deepcopy(game)
        ok=temp_g._Game__move(m.position, m.move,self.player_id)
        if ok:
            temp_g.current_player_idx=1-temp_g.current_player_idx
            t=self.__alphabeta_node(temp_g, t, self.depth, 1-self.player_id)
            outs[m]=t

    return max(outs, key=outs.get)

def __alphabeta_node(self, game:Game, comparison_val, n=4, turn=1) -> int:
        global moves
        if n<0:
            return loss_v1(game)

        outs=[]
        for m in moves:
            temp_g=deepcopy(game)

            assert turn==temp_g.get_current_player(),"Wrong turn"
            ok=temp_g._Game__move(m.position, m.move,turn)

            t=None
            if ok:
                temp_g.current_player_idx=1-temp_g.current_player_idx
                t=self.__alphabeta_node(temp_g, t, n-1, 1-turn)
                if comparison_val is None:
                    outs.append(t)
                elif turn!=self.player_id:
                    if t<comparison_val:
                        return t
                    else:
                        outs.append(t)
                elif turn==self.player_id:
                    if t>comparison_val:
                        return t
                    else:
                        outs.append(t)
                #print(outs)
```

```
        if len(outs)==0:
            print(f"Hit bottom at {n}")
            return -1

        if turn==1-self.player_id:
            return min(outs)
        elif turn==self.player_id:
            return max(outs)
```

The performance is slightly better but not as fast as montecarlo. Again is a recursive function that takes in a comparison value that is used for the pruning. If it's a node belonging to the agent and the value of the children node is larger than the comparison value then it returns without further calculations, if it's an adversary node and the value of the children node is smaller than the comparison value then it returns without further calculations.

Montecarlo Tree Search

For this algorithm I needed an additional class to handle the node, thus it's not recursive.

Every node is evaluated according to the following function:

```
def calc_value_UCT(self, c=2, with_heuristic=False):
        if self.n==0:
            print("Shouldn't happen")
            return
        N=self.n
        if self.parent!=None:
            N=self.parent.n

        val=self.w/self.n + sqrt(c*log(N)/self.n)
        if with_heuristic:
            val+= loss_v1(self.game_state)/5
        return val
```

That takes into account wins over visits but also makes sure that there is an equilibrium in how often nodes are visited, to achieve this feature the second term $\sqrt{\frac{c\log(N)}{n}}$ $\sqrt{nc\log(N)}$ does the trick, if the node has been visited few times with respect to the other the second term will increase the value otherwise will tend to zero

There are essntially 4 steps for the algorithm:

- Search the node with highest UCT value
- Expand it
- Rollout (play some simulations)
- Backpropagate (update all the parent nodes with relative wins and simulations)

For the expansion the agent chooses a random move up to a certain depth that in my case was 100 and plays a certain number of plays which in my case was 5. Depth and Number of plays were working for me to achieve a win rate above 90%.

Advantage Actor Critic

A network is responsible for learning the expected reward and another one is responsible for calculating the probabilities of choosing a move, the advantage (difference between the expected value and the actual reward value) is used to update both the critic and the actor. It's an on policy algorithm that learns as it plays. The implementation of the algorithm didn't work well for this task.

Deep Q Learning

It calculates at the beginning all the legal moves, which are 44. The network is the following

```
class QNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.seq=nn.Sequential(nn.Linear(75,128),
                               nn.ReLU(),
                               nn.Linear(128,64),
                               nn.ReLU(),
                               nn.Linear(64,44))
```

```
    def forward(self, data) -> torch.Tensor:
        return self.seq(data)
```

Each of the 44 values is the expected return value when choosing that action. Certain action must be masked according to the state.

The state for simplicity is encoded in one-hot encoding and then unrolled to be fed to the network (hence the 75 inputs).

The target network used to estimate the bellman equation is slowly updated to follow the agent at each train step.

An Epsilon-Greedy function is used to explore random moves at the beginning of the training and shifting towards an exploiting behaviour in later epochs.

With a discount of .9 and 5000 episodes it reached a 93% percentage of wins but with an incredible speed compared to other methods (about 200 games per second).