Report Computational Intelligence

Berardo Nicholas s319439

Lab 1 -- Set Covering

```
import numpy as np
from random import random
from functools import reduce
from queue import PriorityQueue
```

What is set covering

The goal of set covering is to find all the sets that covers all the values. We have `NUM_SETS` that is the number of sets and `PROBLEM_SIZE` that is the lenght of the values of each set. The value is choosen between `True` and `False`.

```
PROBLEM_SIZE = 8
NUM_SETS = 10
```

```
sets = [np.array([random() < .3 for _ in range(PROBLEM_SIZE)]) for _ in
range(NUM_SETS)]
assert np.all(reduce(np.logical_or,[sets[i] for i in range(NUM_SETS)])),
"Not solvable"
```

State

The state , for exampe `state = ({1,3,5}, {0,2,4,6,7})` is composed of 2 parts:

- The first part is the taken sets

- The second part is the not taken sets

The goal is to cover all the sets, but with the minimum number of taken sets

```
state = ({1,3,5}, {0,2,4,6,7})
#We took 1,3,5 and not taken 0,2,4,6,7, we'll rapresent the states in this
way, 2 sets.
```

Functions

- `goal_check(state)`: returns *True* only if the first set of the state contains all True, meaning that the sets are all covered.
- `distance(state)`: returns the number of sets that have to be covered.
- `h(state)`: this is the heristic function.
    - `largest_set_size` is the set with the max number of *True*.
    - `missing_size` is the number of uncovered values to get to the goal (everything covered). At the end the heuristic tells us how many sets we need at least to get the job done. For example if `larget_set_size` is 4 and `missing_size` is 5 we'll need at least 2 sets for compleating the covering. (we have that the max set is 4, to get to 5 we'll need at least 2 sets)
- `actual_cost(state)`: return the actual cost, that is the lenght of the taken sets.
- `f(n)`: this is the A* function. The goal of A* is to look at the actual cost (the past) but also have an heuristic about the future, given the actual node.

```
def goal_check(state):
    return np.all(reduce(np.logical_or,[sets[i] for i in
state[0]],np.array([False for _ in range(PROBLEM_SIZE)])))
#the reduce does the OR operation on the sets[i] for i in state[0], so the
elements taken!!. Then return the AND between the elementes returned
# by the reduce (by the np.all()).

def distance(state):
    return PROBLEM_SIZE - sum(reduce(np.logical_or,[sets[i] for i in
state[0]],np.array([False for _ in range(PROBLEM_SIZE)])))
#This function  return the distance of the state from the goal. That's the
number of false we still have and
# that has to be covered. The sum returs us the number of true value.

def covered(state):
    return reduce(
        np.logical_or,
        [sets[i] for i in state[0]],
        np.array([False for _ in range(PROBLEM_SIZE)]),
    )

def h(state):
    largest_set_size = max(sum(s) for s in sets)
    missing_size = PROBLEM_SIZE - sum(covered(state))
    optimistic_estimate = np.ceil(missing_size / largest_set_size)
    return optimistic_estimate
```

```
def actual_cost(state):
    return len(state[0]) #The actual cost is the number of elements I have
in the first set

def f(n):
    return actual_cost(n) + h(n) #This is the A* function
```

Code

We use a `PriorityQueue` where the "key" is the f(n) !! So when we are doing `_,state = frontier.get()` we'll get the state with the lowest `f(state)`

```
frontier = PriorityQueue()
initial_state = (set(),set(range(NUM_SETS))) #everithing not taken !
frontier.put((f(initial_state),initial_state))

_,state = frontier.get()
counter =0
while not goal_check(state):
    for a in state[1]: #in state[1] I have all the elements that I didn't
take
        counter+=1
        new_state = (state[0] | {a}, state[1] - {a}) #The | is UNION, - is
DIFFERENCE
        frontier.put((f(new_state),new_state))
    _,state = frontier.get()
```

```
print(counter)
state
```

```
112
```

```
({0, 2, 6}, {1, 3, 4, 5, 7, 8, 9})
```

Lab 2 -- ES

Solution and Notebook made by Riccardo Cardona (https://github.com/Riden15) and Nicholas Berardo (https://github.com/Niiikkkk)

Task

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

The goal of the game is to **avoid** taking the last object.

- Task2.1: An agent using fixed rules based on *nim-sum* (i.e., an *expert system*)
- Task2.2: An agent using evolved rules using ES

Instructions

- Create the directory `lab2` inside the course repo
- Put a `README.md` and your solution (all the files, code and auxiliary data if needed)

Notes

- Working in group is not only allowed, but recommended (see: Ubuntu and Cooperative Learning). Collaborations must be explicitly declared in the `README.md`.
- Yanking from the internet is allowed, but sources must be explicitly declared in the `README.md`.

```
import logging
from pprint import pprint, pformat
from collections import namedtuple
import random
from copy import deepcopy
```

The *Nim* and *Nimply* classes

```
Nimply = namedtuple("Nimply", "row, num_objects")
```

`Nim` is the class of the Nim game. In the *init* function we set the rows, that is an array like [1,3,5,7,...]. The *nimming* method first checks if the move can be done, then if possible subrtact the num_objects to the row specified in the move. The *reset* method is used for training.

```
class Nim:
    def __init__(self, num_rows: int, k: int = None) -> None:
        self.initial_rows = [i * 2 + 1 for i in range(num_rows)]
        self._rows = self.initial_rows.copy()
        self._k = k

    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)

    def nimming(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 reset(self):
        self._rows = self.initial_rows.copy()
```

Sample (and silly) startegies

```
def pure_random(state: Nim) -> Nimply:
    """A completely random move"""
    #takes a row r if the value of that row c is > 0
    row = random.choice([r for r, c in enumerate(state.rows) if c > 0])
    #take a random value from 1 to the value of the row
    num_objects = random.randint(1, state.rows[row])
    #subtract it
    return Nimply(row, num_objects)
```

```
def gabriele(state: Nim) -> Nimply:
    """Pick always the maximum possible number of the lowest row"""
    possible_moves = [(r, o) for r, c in enumerate(state.rows) for o in
range(1, c + 1)]
    return Nimply(*max(possible_moves, key=lambda m: (-m[0], m[1])))
```

```
import numpy as np

#nim_sum mi restituisce XOR in binario delle varie righe che ci sono.
#Se ho 1 e 3, quindi 2 righe:
# 1 -> 01
# 3 -> 11
# 1 XOR 3 = 10 -> 2
def nim_sum(state: Nim) -> int:
    # {c:032b} mi trasforma il valore di state.rows in binario su 32 bit
    tmp = np.array([tuple(int(x) for x in f"{c:032b}") for c in
state.rows])
    #Qua è come se facessi XOR, la posso fare con una somma
    xor = tmp.sum(axis=0) % 2
    return int("".join(str(_) for _ in xor), base=2)
```

```
def analize(raw: Nim) -> dict:
    cooked = dict()
    cooked["possible_moves"] = dict()
    for ply in (Nimply(r, o) for r, c in enumerate(raw.rows) for o in
range(1, c + 1)):
        tmp = deepcopy(raw)
        tmp.nimming(ply)
        cooked["possible_moves"][ply] = nim_sum(tmp)
    return cooked


def optimal(state: Nim) -> Nimply:
    analysis = analize(state)
    logging.debug(f"analysis:\n{pformat(analysis)}")
    spicy_moves = [ply for ply, ns in analysis["possible_moves"].items()
if ns != 0]
    if not spicy_moves:
        spicy_moves = list(analysis["possible_moves"].keys())
    ply = random.choice(spicy_moves)
    return ply
```

Adaptive

- `NimAgent` is the class that represent an individual. In our problem, the individual is an array of number between 0 and 1 where each number indicates the probability of making a move in the nim row equivalent to the index of the single number in the array.
- `make_move`: is the function that, given the `state` of the game, first create a possible move for each row of the game board and then take the best one using the individual's odds.

```
class NimAgent:
    def __init__(self, individual):
        self.individual = individual

    def __str__(self) -> str:
        return str(self.individual)

    def make_move(self, state) -> Nimply:
        #(row,obj)
        moves = [
            (row, min(state.rows[row], max(1, int(self.individual[row] *
state.rows[row]))))
            for row in range(len(state.rows)) if state.rows[row] > 0]

        # Select the move with the highest preference, the highest
preference is the one with the highest genome value.
        # Self.individual[x[0]] indicates the probability associated with
the row of the game board x[0] that we want to modify
        chosen_move = max(moves, key=lambda x: self.individual[x[0]])
```

```
        # Return the move
        return Nimply(*chosen_move) #the * is used for unpacking,
chosen_move is an array!!! We want only the element inside
```

Evolutionary algorithm functions

- `reproduce`: function that creates new individuals with `mutation` and `crossover` based on the `mutation_probability`.
- `mutation`: function that modifies every probability in the individual selected by the `reproduce` function. It modifies the values with a Gaussian distribution making sure that the genome is within 0 and 1 after the mutation.
- `crossover`: It creates a child genome by randomly selecting each gene from either parent with equal probability.
- `replacement`: This function receives as parameters the population, the new individuals crated by the `reproduce` function and the `fitness_score` that indicates the times one individual wins against the optimal algorithm. In this function we sort the population based on the `fitness_score` and replace the worst individuals. In the end, half of the population will be changed and the other half will be preserved.

```
def reproduce(selected_agents, mutation_rate):
    new_population = []

    while len(new_population) < len(selected_agents):
        if random.random() < mutation_rate:
            parent1 = random.sample(selected_agents, 1)[0]
            child_genome = mutation(parent1.individual, mutation_rate)
        else:
            parent1, parent2 = random.sample(selected_agents, 2)
            child_genome = crossover(parent1.individual,
parent2.individual)
        new_population.append(NimAgent(child_genome))
    return new_population

def mutation(genome, mutation_rate, mu=0, sigma=1):
    for prob in range(len(genome)):
        if random.random() < mutation_rate:
            mutation = random.gauss(mu,sigma)
            genome[prob] = min(max(genome[prob] + mutation, 0), 1)
    return genome

def crossover(genome1, genome2):
    child_genome = [g1 if random.random() > 0.5 else g2 for g1, g2 in
zip(genome1, genome2)]
    return child_genome

def replacement(population, new_population, fitness_scores):
    sorted_population = sorted(zip(population, fitness_scores), key=lambda
x: x[1], reverse=True)
```

```
    survivors = sorted_population[:len(population) - len(new_population)]
    return [agent for agent, score in survivors] + new_population
```

Populations functions

- `initialize_population`: this function creates a population of size `population_size` composed of elements of type `NimAgent`. This element is composed of an array named `individual` with size `genome_length`. Each value of the array is a random number between 0 and 1 that represents the probability of making a move in the row equivalent to the index of the element in the array.
- `evaluate_population`: this function evaluates the population with the number of wins they can achieve against the optimal algorithm. In a game therefore our algorithm will be based on making a move on the line with greater probability written in an `individual`. We pass as input the `population` that is the array of `NimAgent`, `nim` is the status of the game and `num_games` represent the number of matches that the population has to do against the optimal algorithm.
- `select_agents`: this function implements the logic to select the fittest agents and to do that we use a simple tournament selection. it takes 2 random participants from the population and selects the winner based on how many times this individual wins against the optimal algorithm i.e., the `fitness_score`. We repeat this process until we have half of the `population_size`.

```
def initialize_population(pop_size, genome_length):
    population = [NimAgent([random.random() for _ in
range(genome_length)]) for _ in range(pop_size)]
    return population

def evaluate_population(population : [NimAgent], nim: Nim, num_games:
int):
    wins = []
    for individual in population:
        strategy = (optimal,individual.make_move)
        win = 0
        for _ in range(num_games):
            nim.reset()
            player = 0
            while nim:
                if player == 1:
                    ply = strategy[player](nim)
                else:
                    ply = strategy[player](nim)
                nim.nimming(ply)
                player = 1 - player
            if player == 1:
                win+=1
        wins.append(win)
    return wins

def select_agents(population, fitness_scores):
    selected = []
    while len(selected) < len(population) // 2:
```

```
            participant = random.sample(list(zip(population, fitness_scores)),
2)
            winner = max(participant, key=lambda x: x[1])
            selected.append(winner[0])
        return selected
```

Evolution Strategy

Problem parameters:

- `nim`: Object of type Nim with the status of the game
- `generations`: number of generations of population to create
- `population_size`: size of the population
- `initial_mutation_rate`: probability to do mutation
- `wins_goal`: number of victories to be achieved to finish the creation of new generations first
- `num_games`: number of games against the optimal algorithm to adjust the fitness scores
- `mutation_probability`: probability to do mutation instead of crossover

In the `evolutionary_strategy` function, we implement the logic of the Evolutionary Algorithm. It returns the population with the best individual found.

```
GENERATIONS = 50
POPULATION_SIZE = 10
WINS_GOAL = 90
NUM_GAMES = 100
MUTATION_PROBABILITY = 0.3
```

```
def evolutionary_strategy(nim, generations, population_size, wins_goal,
num_games, mutation_probability):
    population = initialize_population(population_size, len(nim.rows))
    best_individual = None
    best_fitness = -1

    for generation in range(generations):
        fitness_scores = evaluate_population(population, nim, num_games)

        # The best score is halved to report the number of wins, since
each win is worth double points in the scoring system.
        print(f"Generation {generation}: Best score {max(fitness_scores)}
wins")

        # Check for termination condition (e.g., a perfect score)
        if max(fitness_scores) >= wins_goal:
            print("Stopping early, reached perfect score!")
            break

        # Selection
        selected_agents = select_agents(population, fitness_scores)
```

```
        # Reproduction
        new_population = reproduce(selected_agents, mutation_probability)

        # Replacement
        population = replacement(population, new_population,
fitness_scores)

        # Check if the new best individual is found
        max_fitness = max(fitness_scores)
        if max_fitness > best_fitness:
            best_fitness = max_fitness
            best_individual_index = fitness_scores.index(max_fitness)
            best_individual = population[best_individual_index]

    return population, best_individual
```

Code test

```
nim = Nim(5)
pop, best_ind =
evolutionary_strategy(nim,GENERATIONS,POPULATION_SIZE,WINS_GOAL,NUM_GAMES,
MUTATION_PROBABILITY)
```

```
Generation 0: Best score 41 wins
Generation 1: Best score 32 wins
Generation 2: Best score 36 wins
Generation 3: Best score 37 wins
Generation 4: Best score 38 wins
Generation 5: Best score 42 wins
Generation 6: Best score 39 wins
Generation 7: Best score 48 wins
Generation 8: Best score 38 wins
Generation 9: Best score 41 wins
Generation 10: Best score 35 wins
Generation 11: Best score 43 wins
Generation 12: Best score 38 wins
Generation 13: Best score 36 wins
Generation 14: Best score 39 wins
Generation 15: Best score 39 wins
Generation 16: Best score 44 wins
Generation 17: Best score 40 wins
Generation 18: Best score 38 wins
Generation 19: Best score 44 wins
Generation 20: Best score 52 wins
Generation 21: Best score 40 wins
Generation 22: Best score 47 wins
Generation 23: Best score 47 wins
```

```
Generation 24: Best score 40 wins
Generation 25: Best score 46 wins
Generation 26: Best score 47 wins
Generation 27: Best score 48 wins
Generation 28: Best score 56 wins
Generation 29: Best score 53 wins
Generation 30: Best score 43 wins
Generation 31: Best score 45 wins
Generation 32: Best score 50 wins
Generation 33: Best score 48 wins
Generation 34: Best score 46 wins
Generation 35: Best score 48 wins
Generation 36: Best score 50 wins
Generation 37: Best score 49 wins
Generation 38: Best score 51 wins
Generation 39: Best score 51 wins
Generation 40: Best score 47 wins
Generation 41: Best score 46 wins
Generation 42: Best score 57 wins
Generation 43: Best score 66 wins
Generation 44: Best score 59 wins
Generation 45: Best score 59 wins
Generation 46: Best score 56 wins
Generation 47: Best score 58 wins
Generation 48: Best score 60 wins
Generation 49: Best score 62 wins
```

Oversimplified match

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

strategy = (optimal, best_ind.make_move)

nim = Nim(5)
logging.info(f"init : {nim}")
player = 0
while nim:
    if player == 1 :
        ply = strategy[player](nim)
    else:
        ply = strategy[player](nim)
    logging.info(f"ply: player {player} plays {ply}")
    nim.nimming(ply)
    logging.info(f"status: {nim}")
    player = 1 - player
logging.info(f"status: Player {player} won!")
```

```
INFO:root:init : <1 3 5 7 9>
INFO:root:ply: player 0 plays Nimply(row=3, num_objects=2)
INFO:root:status: <1 3 5 5 9>
INFO:root:ply: player 1 plays Nimply(row=2, num_objects=5)
INFO:root:status: <1 3 0 5 9>
INFO:root:ply: player 0 plays Nimply(row=4, num_objects=4)
INFO:root:status: <1 3 0 5 5>
INFO:root:ply: player 1 plays Nimply(row=3, num_objects=5)
INFO:root:status: <1 3 0 0 5>
INFO:root:ply: player 0 plays Nimply(row=4, num_objects=1)
INFO:root:status: <1 3 0 0 4>
INFO:root:ply: player 1 plays Nimply(row=0, num_objects=1)
INFO:root:status: <0 3 0 0 4>
INFO:root:ply: player 0 plays Nimply(row=1, num_objects=3)
INFO:root:status: <0 0 0 0 4>
INFO:root:ply: player 1 plays Nimply(row=4, num_objects=1)
INFO:root:status: <0 0 0 0 3>
INFO:root:ply: player 0 plays Nimply(row=4, num_objects=2)
INFO:root:status: <0 0 0 0 1>
INFO:root:ply: player 1 plays Nimply(row=4, num_objects=1)
INFO:root:status: <0 0 0 0 0>
INFO:root:status: Player 0 won!
```

LAB9 -- ES

Solution and Notebook made by Riccardo Cardona (https://github.com/Riden15) and Nicholas Berardo (https://github.com/Niiikkkk)

Write a local-search algorithm (eg. an EA) able to solve the *Problem* instances 1, 2, 5, and 10 on a 1000-loci genomes, using a minimum number of fitness calls. That's all.

Deadlines:

- Submission: Sunday, December 3 (CET)
- Reviews: Sunday, December 10 (CET)

Notes:

- Reviews will be assigned on Monday, December 4
- You need to commit in order to be selected as a reviewer (ie. better to commit an empty work than not to commit)

```
from random import choices,randint

import lab9_lib
```

Problem

First of all the lab9_lib should be considered as a black box. The goal is to obtain a good fitness value without even knowing what problem is. Below we can see the fitness of random 50 bit.

```
fitness = lab9_lib.make_problem(10)
for n in range(10):
    ind = choices([0, 1], k=50)
    print(f"{''.join(str(g) for g in ind)}: {fitness(ind):.2%}")

print(fitness.calls)
```

```
10011001000001011011111101011001000100011100110111: 7.33%
11010000100011101101011100000110011111011001010100: 7.33%
01000010000111000111100011101001001100010000110000: 17.56%
11111000110101010111000100100100010001100010100011: 23.56%
10010000110100100011010001110101011110011011111000: 7.33%
11000001001111110100111100010111001001000010100000: 23.34%
00101100101100110101001100011101011110011001110101: 23.34%
11111011111011110000100100000101100010100010101100: 7.33%
01010100000100001100001010101010110000111101100011: 17.56%
10001010011110011101100101011100001001011100001000: 15.34%
10
```

EA

The genome here is a tuple composed of (loci,fitness value).

Methods

- `mutation()` flips a random bit from the genome
- `one_cut_xover()` does the one cut crossover (select a point, then take everything from the start to that point from parent1, from the point to the end from parent2)
- `xover` does the crossover for each bit in the genome from parent1 and parent2

```
import random

def mutation(genome):
    index = randint(0,len(genome[0])-1)
    genome[0][index] = 1-genome[0][index]
    return genome[0]

def one_cut_xover(ind1, ind2):
    cut_point = randint(0, len(ind1[0]))
```

```
    offspring = ind1[0][:cut_point]+ind2[0][cut_point:]
    return offspring

def xover(genome1, genome2):
    child_genome = [g1 if random.random() > 0.5 else g2 for g1, g2 in
zip(genome1[0], genome2[0])]
    return child_genome
```

- `select_parent` take one individual as champion. *best_parents* is composed of the first population/2 best individual(the ones with high fitness). From this extract 1/10 and pick the champion from this pool.
- `init_population` init the population with *n_individual*. The single individual is a tuple (genome,fitness), where genome is the random loci and fitness is the fitness of the genome

```
from random import choice

def select_parent(population,fitness):
    best_parents = sorted(population,key= lambda i:i[1],reverse=True)
[:int(len(population)/2)]
    pool = [choice(best_parents) for _ in
range(int(len(best_parents)/10))]
    champion = max(pool, key=lambda i: i[1])
    return champion

def init_population(n_individual,length,fitness):
    pop = []
    for _ in range(n_individual):
        ind = (choices([0, 1], k=length))
        pop.append((ind,fitness(ind)))
    return pop
```

- `gen_new_population` generates the new population. We have a *mutation_prob* to do the mutation, otherwise we do the xover. Both uses the *select_parent* function to select the individual to mutate or to apply xover
- `replacement` combines the new population and the old population, then select the best ones based on the fitness.

```
def
gen_new_population(n_new_indivdual,mutation_prob,old_population,fitness):
    new_individual = []
    for _ in range(n_new_indivdual):
        if random.random() < mutation_prob:
            old_ind = select_parent(old_population,fitness)
            tmp = mutation(old_ind)
        else:
            old_ind = select_parent(old_population,fitness)
            old_ind_2 = select_parent(old_population,fitness)
            tmp = xover(old_ind,old_ind_2)
```

```
            new_individual.append((tmp,fitness(tmp)))
        return new_individual

    def replacement(new_pop,old_pop,fitness):
        tmp_pop = new_pop + old_pop
        sorted_pop = sorted(tmp_pop,key= lambda i:i[1],reverse=True)
        return sorted_pop[:len(new_pop)]
```

Train

We train for 1,2,5,10 problem size and report the results.

- We have 100 genome in the population
- The genomes has 1000 lenght
- We train for 300 generation for each problem size

```
N_INDV = 100
LENGTH_INDV = 1000
GENERATION = 300

problem_size = [1,2,5,10]

for ps in problem_size:
    fit = lab9_lib.make_problem(ps)
    pop = init_population(N_INDV,LENGTH_INDV,fit)
    best = 0
    n_calls = 0
    gen = 0
    for g in range(GENERATION):
        #if g%10 == 0:
        #    avg_fit = sum(list(map(lambda i:i[1],pop)))/len(pop)
        #    print(pop[0][1], avg_fit)
        new_pop = gen_new_population(N_INDV,0.1,pop,fit)
        pop = replacement(new_pop,pop,fit)
        if pop[0][1] > best:
            best = pop[0][1]
            n_calls = fit.calls
            gen = g
    print(f"Problem size: {ps}")
    print(f"Best fitness: {best}")
    print(f"Fitness calls: {n_calls}")
    print(f"Generation: {gen}")
    print(f"Population size: {N_INDV}")
```

```
Problem size: 1
Best fitness: 0.978
Fitness calls: 29500
Generation: 293
Population size: 100
```

```
Problem size: 2
Best fitness: 0.732
Fitness calls: 30000
Generation: 298
Population size: 100
Problem size: 5
Best fitness: 0.4273
Fitness calls: 29300
Generation: 291
Population size: 100
Problem size: 10
Best fitness: 0.219013335
Fitness calls: 30100
Generation: 299
Population size: 100
```

LAB10 -- RL

Use reinforcement learning to devise a tic-tac-toe player.

Deadlines:

- Submission: Sunday, December 17 (CET)
- Reviews: Dies Natalis Solis Invicti (CET)

Notes:

- Reviews will be assigned on Monday, December 4
- You need to commit in order to be selected as a reviewer (ie. better to commit an empty work than not to commit)

```
import numpy as np
```

State of the board

The board is composed of 1 if p1 is playing, -1 if p2 is playing or 0 if the cell is empty. The winner gets a reward of 1, meanwhile the loser gets a rewoard of 0. If there's a tie, we give the RL player (the p1) a reward of 0.1 because we don't want ties!

Methods

- `__init__` defines the board and sets the player
- `available_positions` returns the available positions, the positions in which no one played
- `make_move` takes a position as input, check if it's valid and if it is the current_player makes the move in that position

- `get_hash` return the "hash" of the table, beacuse than we have to save it, and we can't save a np array
- `check_winner` checks if the game has ended and who's the winner in all possible directions
- `reward` give the reward to the winner/loser
- `reset` resets the game, this is needed for the training
- `show_board` prints the board in a pretty way
- `train` make p1 play against p2 for many rounds, than at every move check if someone won, gives the reward and reset the game for another round. Here we also add the state to the players, so they can save it and use it with RL
- `test` makes p1 play againt p2 for one round, check the winner at every move and returns the winner, without setting any state or giving any reward

```
class State:
    def __init__(self,p1,p2):
        self.board = np.zeros((3,3))
        self.p1 = p1
        self.p2 = p2
        self.isEnd = False
        self.current_player = 1 #1 is p1, -1 is p2

    def available_positions(self):
        pos = []
        for i in range(3):
            for j in range(3):
                if self.board[i,j] == 0:
                    pos.append((i,j))
        return pos

    def make_move(self, position):
        if position not in self.available_positions():
            return None
        self.board[position] = self.current_player
        self.current_player = self.current_player*-1

    def getHash(self):
        self.boardHash = str(self.board.reshape(3 * 3))
        return self.boardHash

    def check_winner(self):
        #check if rows contains 3 or -3 (some one win)
        for i in range(3):
            if sum(self.board[i,:]) == 3:
                self.isEnd = True
                return 1 #player 1 won
        for i in range(3): #loop on the rows
            if sum(self.board[i,:]) == -3:
                self.isEnd = True
                return -1 #player 2 won

        #check if col contains 3 or -3
        for i in range(3):
```

```
            if sum(self.board[:,i]) == 3:
                self.isEnd = True
                return 1
        for i in range(3):
            if sum(self.board[:,i]) == -3:
                self.isEnd = True
                return -1

        #check diagonal win
        diag_sum = sum([self.board[i,i] for i in range(3)])
        if diag_sum == 3:
            self.isEnd= True
            return 1
        if diag_sum == -3:
            self.isEnd = True
            return -1

        diag_sum = sum([self.board[i,3-i-1] for i in range(3)])
        if diag_sum == 3:
            self.isEnd= True
            return 1
        if diag_sum == -3:
            self.isEnd = True
            return -1

        #here no one won..
        if len(self.available_positions())==0 :
            self.isEnd = True
            return 0 #no one won

        return None #Here there are still moves, so keep playing !!!

    def reward(self):
        result = self.check_winner()

        if result == 1:
            self.p1.give_rew(1) #player 1 won, so give 1 reward
            self.p2.give_rew(0)
        elif result == -1:
            self.p1.give_rew(0)
            self.p2.give_rew(1)
        else:
            self.p1.give_rew(0.1) #give a less reward because we don't
want ties
            self.p2.give_rew(0.5)

    def reset(self):
        self.board = np.zeros((3, 3))
        self.boardHash = None
        self.isEnd = False
        self.playerSymbol = 1

    def showBoard(self):
        # p1: x  p2: o
```

```
        for i in range(0, 3):
            print('-------------')
            out = '| '
            for j in range(0, 3):
                if self.board[i, j] == 1:
                    token = 'x'
                if self.board[i, j] == -1:
                    token = 'o'
                if self.board[i, j] == 0:
                    token = ' '
                out += token + ' | '
            print(out)
        print('-------------')

    def train(self, rounds=100):
        for i in range(rounds):
            if i % 1000 == 0:
                print("Rounds {}".format(i))
            while not self.isEnd:
                # Player 1
                positions = self.available_positions()
                p1_action = self.p1.chooseAction(positions, self.board,
self.current_player)
                # take action and upate board state
                self.make_move(p1_action)
                board_hash = self.getHash()
                self.p1.addState(board_hash)
                # check board status if it is end

                win = self.check_winner()
                if win is not None: #It returns None only when no one
finished or tied.
                    # self.showBoard()
                    # ended with p1 either win or draw
                    self.reward() #send rewards to the players, the game
has ended
                    self.p1.reset()
                    self.p2.reset()
                    self.reset()
                    break

                else:
                    # Player 2
                    positions = self.available_positions()
                    p2_action = self.p2.chooseAction(positions,
self.board, self.current_player)
                    self.make_move(p2_action)
                    board_hash = self.getHash()
                    self.p2.addState(board_hash)

                    win = self.check_winner()
                    if win is not None:
                        # self.showBoard()
                        # ended with p2 either win or draw
```

```
                        self.reward()
                        self.p1.reset()
                        self.p2.reset()
                        self.reset()
                        break

    def test(self):
        while not self.isEnd:
            # Player 1
            positions = self.available_positions()
            p1_action = self.p1.chooseAction(positions, self.board,
self.current_player)
            # take action and upate board state
            self.make_move(p1_action)
            #self.showBoard()
            # check board status if it is end
            win = self.check_winner()
            if win is not None: #if win not None means some one win or tie
                return win

            else:
                # Player 2
                positions = self.available_positions()
                p2_action = self.p2.chooseAction(positions, self.board,
self.current_player)

                self.make_move(p2_action)
                #self.showBoard()
                win = self.check_winner()
                if win is not None:
                    return win
```

RLPlayer

The RL player if composed of an array of states and a dictionary of state_values. The action is choosen as follow:

- 30% of the time a random action is choosen
- 70% of the time we pick the action with the highest value

The give_rew() function iterates over the array of states and applies the following formula:

- V(t) = V(t) + lr * (gamma*V(t+1) - V(t))

Methods

- `init` sets the RL player, the array of states and the state_values dict, and some parameters used in the following
- `getHash` returns the hash of the table (we can't save the table as a np array, it's better to save it as a string and use it as a key in the states_value dictionary)

- `addState` add the state to the states array (this is called by the training function above)
- `chooseAction` this function return the action to perform in the board. It receives the available_positions, the current_board and the symbol (the player making the move). With a probability of 0.3 (*exp_rate*) does exploration, so picks a random move. For the rest 70% picks the best move among the available. The best move is the one with the highest value in the states_value dictionary.
- `reset` reallocate the array of states
- `give_rew` loops trought the reverse state array and update the values in the dictionary with the formula:
    - V(t) = V(t) + lr * (gamma*V(t+1) - V(t))

```
class RLPlayer:
    def __init__(self, name, exp_rate = 0.3):
        self.name = name
        self.states = []  # record all positions taken
        self.lr = 0.2
        self.exp_rate = exp_rate
        self.decay_gamma = 0.9
        self.states_value = {}  # state -> value

    def getHash(self, board):
        boardHash = str(board.reshape(3*3))
        return boardHash

    def addState(self, state):
        self.states.append(state)

    def chooseAction(self, positions, current_board, symbol):
        """Return a random action (P = 0.3) or the action with max value
(P = 0.7)"""
        if np.random.uniform(0, 1) <= self.exp_rate: # Do exploration,
take random
            # take random action
            idx = np.random.choice(len(positions))
            action = positions[idx]
        else: #Here do exploitation, take the action that has highest
value
            value_max = -999
            for p in positions:
                next_board = current_board.copy() #create a tmp board
                next_board[p] = symbol #do the action
                next_boardHash = self.getHash(next_board) #get the hash
                value = 0 if self.states_value.get(next_boardHash) is None
else self.states_value.get(next_boardHash)
                # print("value", value)
                if value >= value_max: #find the action that has max
value.
                    value_max = value
                    action = p
        return action
```

```
    def reset(self):
        self.states = []

    def give_rew(self, reward):
        #At the end of the game i'll get a reward. The iterate on the
states in reverse.
        # set the value of the state to 0 if not existing, otherwise
update it with the reward.
        for st in reversed(self.states):
            if self.states_value.get(st) is None: #if the state doesn't
have a value, set it to 0
                self.states_value[st] = 0
            #this is V(t) = V(t) + lr * (gamma*V(t+1) - V(t))
            self.states_value[st] += self.lr * (self.decay_gamma * reward
- self.states_value[st])
            reward = self.states_value[st]
```

Human player

Here we have a human player, so we have an input

```
class HumanPlayer:
    def __init__(self, name):
        self.name = name

    def chooseAction(self, positions,current_board, symbol):
        while True:
            row = int(input("Input your action row:"))
            col = int(input("Input your action col:"))
            action = (row, col)
            if action in positions:
                return action

    # append a hash state
    def addState(self, state):
        pass

    # at the end of game, backpropagate and update states value
    def give_rew(self, reward):
        pass

    def reset(self):
        pass
```

Random player

Here we perform actions randomly, taken from the set of possible actions (positions)

```
class RandomPlayer:
    def __init__(self, name):
        self.name = "random"

    def chooseAction(self, positions,current_board, symbol):
        x = np.random.randint(0,len(positions)-1)
        return positions[x]

    def addState(self,state):
        pass

    def give_rew(self, reward):
        pass

    def reset(self):
        pass

    def give_rew(self,rew):
        pass
```

Training

Here we train using 2 RL player, than test it with a random player

```
p1 = RLPlayer("computer")
p2 = RLPlayer("computer")

st = State(p1,p2)
st.train(100000)
```

```
Rounds 0
Rounds 1000
Rounds 2000
Rounds 3000
Rounds 4000
Rounds 5000
Rounds 6000
Rounds 7000
Rounds 8000
Rounds 9000
Rounds 10000
Rounds 11000
Rounds 12000
Rounds 13000
Rounds 14000
Rounds 15000
Rounds 16000
```

```
Rounds 17000
Rounds 18000
Rounds 19000
Rounds 20000
Rounds 21000
Rounds 22000
Rounds 23000
Rounds 24000
Rounds 25000
Rounds 26000
Rounds 27000
Rounds 28000
Rounds 29000
Rounds 30000
Rounds 31000
Rounds 32000
Rounds 33000
Rounds 34000
Rounds 35000
Rounds 36000
Rounds 37000
Rounds 38000
Rounds 39000
Rounds 40000
Rounds 41000
Rounds 42000
Rounds 43000
Rounds 44000
Rounds 45000
Rounds 46000
Rounds 47000
Rounds 48000
Rounds 49000
Rounds 50000
Rounds 51000
Rounds 52000
Rounds 53000
Rounds 54000
Rounds 55000
Rounds 56000
Rounds 57000
Rounds 58000
Rounds 59000
Rounds 60000
Rounds 61000
Rounds 62000
Rounds 63000
Rounds 64000
Rounds 65000
Rounds 66000
Rounds 67000
```

```
Rounds 68000
Rounds 69000
Rounds 70000
Rounds 71000
Rounds 72000
Rounds 73000
Rounds 74000
Rounds 75000
Rounds 76000
Rounds 77000
Rounds 78000
Rounds 79000
Rounds 80000
Rounds 81000
Rounds 82000
Rounds 83000
Rounds 84000
Rounds 85000
Rounds 86000
Rounds 87000
Rounds 88000
Rounds 89000
Rounds 90000
Rounds 91000
Rounds 92000
Rounds 93000
Rounds 94000
Rounds 95000
Rounds 96000
Rounds 97000
Rounds 98000
Rounds 99000
```

Testing

```
p2 = RandomPlayer("Random")
epochs = 1000
st = State(p1,p2)
win_comp = 0

for epoch in range(epochs):
    win = st.test()
    if win == 1:
        win_comp+=1
    st.reset()

perc = win_comp / epochs * 100
perc
```

```
75.3
```

Have fun, play against it

```
p2 = HumanPlayer("human")

st = State(p1, p2)
st.test()
```

Lab10 --BIS -- We also tried to implement QL

```
from collections import defaultdict
import numpy as np
import random
from tqdm.auto import tqdm
```

State

The state is exactly the same as the RL, we don't have the `give_reward` function

```
class State:
    def __init__(self):
        self.board = np.zeros((3,3))
        self.isEnd = False
        self.current_player = 1 #1 is p1, -1 is p2

    def available_moves(self):
        pos = []
        for i in range(3):
            for j in range(3):
                if self.board[i,j] == 0:
                    pos.append((i,j))
        return pos

    def make_move(self, position):
        if position not in self.available_moves():
            return None
        self.board[position] = self.current_player
        self.current_player = self.current_player*-1

    def getHash(self):
        self.boardHash = str(self.board.reshape(3 * 3))
        return self.boardHash
```

```
    def check_winner(self):
        #check if rows contains 3 or -3 (some one win)
        for i in range(3):
            if sum(self.board[i,:]) == 3:
                self.isEnd = True
                return 1 #player 1 won
        for i in range(3): #loop on the rows
            if sum(self.board[i,:]) == -3:
                self.isEnd = True
                return -1 #player 2 won

        #check if col contains 3 or -3
        for i in range(3):
            if sum(self.board[:,i]) == 3:
                self.isEnd = True
                return 1
        for i in range(3):
            if sum(self.board[:,i]) == -3:
                self.isEnd = True
                return -1

        #check diagonal win
        diag_sum = sum([self.board[i,i] for i in range(3)])
        if diag_sum == 3:
            self.isEnd= True
            return 1
        if diag_sum == -3:
            self.isEnd = True
            return -1

        diag_sum = sum([self.board[i,3-i-1] for i in range(3)])
        if diag_sum == 3:
            self.isEnd= True
            return 1
        if diag_sum == -3:
            self.isEnd = True
            return -1

        #here no one won..
        if len(self.available_moves())==0 :
            self.isEnd = True
            return 0 #no one won

        return None #Here there are still moves, so keep playing !!!

    def reset(self):
        self.board = np.zeros((3, 3))
        self.boardHash = None
        self.isEnd = False
        self.playerSymbol = 1

    def showBoard(self):
        # p1: x  p2: o
        for i in range(0, 3):
```

```
            print('-------------')
            out = '| '
            for j in range(0, 3):
                if self.board[i, j] == 1:
                    token = 'x'
                if self.board[i, j] == -1:
                    token = 'o'
                if self.board[i, j] == 0:
                    token = ' '
                out += token + ' | '
            print(out)
        print('-------------')
```

QL player

The reward() return 1 if the player won, -1 if the player lost and 0 if no one won, or the game is still going

Methods

- `init` initialize the dictionary Q and sets some parameters. The Q table is a dict that has as key the tuple (board,action)
- `chooseAction` again for 30% of the time perform exploration doing a random move, otherwise take the values from the Q table for that board and all the possilbe action that can be done from that state. Then take the action with the highest value
- `addState` add the state and action to the states list
- `reset` resets the states list
- `update_Q` updates the Q table in a similar way done with RL.
    - Q(t) = Q(t) + alpha * (df * Q(t+1) - Q(t))

```
class QL:
    def __init__(self,name,alpha,eps,disc_factor):
        self.Q = defaultdict(lambda: 0.0)
        self.name = name
        self.alpha = alpha
        self.eps = eps
        self.disc_factor = disc_factor
        self.states = []

    def chooseAction(self,board,moves):
        if random.random() < self.eps:
            return random.choice(moves)
        else:
            values = [self.Q[(board,a)] for a in moves]
            max_value = np.max(values)
            if values.count(max_value) > 1: #if there're more than one
actions that has max value take random
                best_move = [i for i in range(len(values)) if values[i] ==
max_value]
                i = random.choice(best_move)
            else:
```

```
                i = values.index(max_value)
            return moves[i]

    def addState(self,state,move):
        self.states.append((state,move))

    def reset(self):
        self.states=[]

    def update_Q(self,reward):
        for st in reversed(self.states):
            current_q_value = self.Q[(st[0], st[1])] # st[0] = board state
st[1] = action
            reward = current_q_value + self.alpha * (self.disc_factor *
reward - current_q_value)
            self.Q[(st[0], st[1])] = reward
```

Random player

Performs random moves

```
class RandomPlayer:
    def __init__(self, name):
        self.name = "random"

    def chooseAction(self, game,positions):
        return random.choice(positions)

    def addState(self,state):
        pass

    def reset(self):
        pass

    def reward(self,rew):
        pass

    def update_Q(self,rew):
        pass
```

Train and Test QL Player

The train code is very similar to the RL train, changes just the update_Q instead of give_rew. Also the test is very similar.

```
def train(game:State, p1: QL, p2: RandomPlayer,epochs = 20000):
    for epoch in tqdm(range(epochs)):
        game.reset()
```

```
        p1.reset()
        p2.reset()
        while game.check_winner() is None:
            #Player 1
            possilbe_moves = game.available_moves()
            move = p1.chooseAction(game.getHash(),possilbe_moves)
            p1.addState(game.getHash(),move)
            game.make_move(move)
            if game.check_winner() is not None:
                if game.check_winner() == 1:
                    p1.update_Q(1) #player 1 won, so give 1 reward
                    p2.update_Q(0)
                elif game.check_winner() == -1:
                    p1.update_Q(0)
                    p2.update_Q(1)
                else:
                    p1.update_Q(0.1) #give a less reward because we don't
want ties
                    p2.update_Q(0.5)
            else:
                #Player 2
                possilbe_moves = game.available_moves()
                move = p2.chooseAction(game.getHash(),possilbe_moves)
                game.make_move(move)
                if game.check_winner() is not None:
                    if game.check_winner() == 1:
                        p1.update_Q(1) #player 1 won, so give 1 reward
                        p2.update_Q(0)
                    elif game.check_winner() == -1:
                        p1.update_Q(0)
                        p2.update_Q(1)
                    else:
                        p1.update_Q(0.1) #give a less reward because we
don't want ties
                        p2.update_Q(0.5)

def test(game, p1, p2):
    while game.check_winner() is None:
        #Player 1
        possilbe_moves = game.available_moves()
        move = p1.chooseAction(game.getHash(),possilbe_moves)
        game.make_move(move)
        if len(game.available_moves()) == 0:
            break
        #Player 2
        possilbe_moves = game.available_moves()
        move = p2.chooseAction(game.getHash(),possilbe_moves)
        game.make_move(move)
    return game.check_winner()
```

```
p1 = QL("QL",0.2,0.2,0.9)
p2 = RandomPlayer("Random")
game = State()

train(game,p1,p2,200000)
game.reset()
```

```
  0%|          | 0/200000 [00:00<?, ?it/s]
```

```
test_loop = 1000
win = 0
tie = 0
for t in range(test_loop):
    w = test(game,p1,p2)
    game.reset()
    if w == 1:
        win +=1
    if w == 0:
        tie +=1
print(f"Win: {win/test_loop*100}")
```

```
Win: 81.6
```

Additional Things

8 - Puzzle

Here's the code for the 8-puzzle game. I wrote the code at the very beginning of the course, then didn't have time to look at it anymore, maybe with another implementation like EA it could work better. Anyway, the code has a error that I can't understand.

```
import numpy as np
from queue import PriorityQueue, LifoQueue, SimpleQueue
```

```
def print_state(state):
    for j in range(3):
        for i in range(3):
            print(f"{state[i+3*j]}",end=" ")
        print("\n")
```

```
def goal_check(state):
    return np.all(state == np.array([1,2,3,4,5,6,7,8,-1]))

def distance_check(state):
    return 9-np.sum(state == np.array([1,2,3,4,5,6,7,8,-1]))
```

State

The state is a tuple composed of the actual state of the board and the numbers that can move (that are near the hole).

```
queue = PriorityQueue() #the priority is the distance from the goal
initial_state = (np.array([2,5,7,1,0,8,9,3,4]),np.array([5,1,8,3]))
queue.put((distance_check(initial_state[0]),initial_state))

_,state = queue.get()
current_state,state_to_move = state
while not goal_check(current_state):
    idx_hole = np.where(current_state==0)[0].item() #finds the index of
the hole
    for i in state_to_move: #for each number that can be moved
        idx_to_change = np.where(current_state==i)[0].item() #finds the
index of the number i
        tmp = current_state.copy()
        tmp[idx_hole] = i
        tmp[idx_to_change] = 0
        tmp_move = [] #now create the set of item that can be moved in the
new state
        if idx_to_change-3>=0:
            tmp_move.append(tmp[idx_to_change-3])

        if idx_to_change+3<3*3:
            tmp_move.append(tmp[idx_to_change+3])

        if idx_to_change > 0 and
idx_to_change>3*np.floor((idx_to_change)/3):
            tmp_move.append(tmp[idx_to_change-1])

        if idx_to_change<8 and
idx_to_change+1<3+3*np.floor((idx_to_change)/3):
            tmp_move.append(tmp[idx_to_change+1])

        new_state = (np.array(tmp),np.array(tmp_move))
        print(distance_check(new_state[0]))

        queue.put((distance_check(new_state[0]),new_state)) #put the new
state in the queue
```

```
    _,state = queue.get()
    current_state,state_to_move = state
```

```
8
9
9
9



---------------------------------------------------------------------------
-

ValueError                                Traceback (most recent call
last)

Cell In[7], line 30
     27     new_state = (np.array(tmp),np.array(tmp_move))
     28     print(distance_check(new_state[0]))
---> 30     queue.put((distance_check(new_state[0]),new_state))
     32 _,state = queue.get()
     33 current_state,state_to_move = state


File
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/queue.py
:150, in Queue.put(self, item, block, timeout)
    148                 raise Full
    149             self.not_full.wait(remaining)
--> 150 self._put(item)
    151 self.unfinished_tasks += 1
    152 self.not_empty.notify()


File
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/queue.py
:236, in PriorityQueue._put(self, item)
    235 def _put(self, item):
--> 236     heappush(self.queue, item)


ValueError: The truth value of an array with more than one element is
ambiguous. Use a.any() or a.all()
```

Halloween

```

```

```
from itertools import product
from random import random, randint, shuffle, seed, choice
import numpy as np
from functools import reduce
from scipy import sparse
```

```
def make_set_covering_problem(num_points, num_sets, density):
    """Returns a sparse array where rows are sets and columns are the
covered items"""
    #seed(num_points*2654435761+num_sets+density)
    sets = sparse.lil_array((num_sets, num_points), dtype=bool)
    for s, p in product(range(num_sets), range(num_points)):
        if random() < density:
            sets[s, p] = True
    for p in range(num_points):
        sets[randint(0, num_sets-1), p] = True
    return sets
```

Halloween Challenge

Find the best solution with the fewest calls to the fitness functions for:

- num_points = [100, 1_000, 5_000]
- num_sets = num_points
- density = [.3, .7]

```
num_points = 100
num_sets = num_points
density = .3
x = make_set_covering_problem(num_points, num_sets, density)
```

Example

initial state = [True, True, False, False, False, True, False, True, True, False] x = with array([list([1, 2, 5, 9]), list([2, 7]), list([1, 3, 6, 8, 9]), list([0, 7, 8]), list([3, 5, 6, 8]), list([2, 9]), list([2, 3, 4, 8]), list([1, 2, 8, 9]), list([0, 6, 7, 9]), list([0, 2, 4, 6, 7])], dtype=object) means we take the first ,the second, 6,8,9 sets

```
initiale_state = [choice([True, False]) for _ in range(num_sets)]
```

```
def goal_check(state):
    cost = sum(state)
    is_valid = set(np.concatenate(x.rows[state])) == set(i for i in
```

```
range(num_points))
    return is_valid,-cost
```

Hill Climbing

The tweak function here flips the 30% of the bits in the state.

Then apply the hill climbing algorithm, that tweaks the state and checks if the new state is better. The problem here is that I could remain stuck in a local optima.

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

```
state = initiale_state
for step in range(100):
    new_state = tweak(state)
    if goal_check(state)<goal_check(new_state):
        state=new_state

#print(f"The final state is: {state}")
#print(f"The sets are: {x.rows[state]}")
valid,cost = goal_check(state)
print(f"The solution is: {valid}. I reached the solution with {-cost}
sets, so I have a cost of {cost}")
print(f"Those are the covered sets: {set(np.concatenate(x.rows[state]))}")
```

```
The solution is: True. I reached the solution with 27 sets, so I have a
cost of -27
Those are the covered sets: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
```

Simulated Annealing

The tweak is again the same as before, but the algorithm is different. Here we are doing a hill climbing but with a probability of accepting a worse solution with a probability $$ p = e^{-{f(s)-f(s')\over t}} $$ where s is the current state, s' is the new state and t is the temperature. The temperature t is schedule to decrese.

When t=0 I won't accept any bad solutions. When t is highet I'll move also to bad solutions, this will add exploration.

```
def tweak(state):
    new_state = state.copy()
    for i in range(int(30*num_points/100)): #change 30% of the points
        index = randint(0, num_points - 1)
        new_state[index] = not new_state[index]
    return new_state
```

```
state = initiale_state
t=num_points
for step in range(num_points):
    new_state = tweak(state)
    oldStateCost = goal_check(state)
    newStateCost = goal_check(new_state)
    p=np.exp(-((oldStateCost[1]-newStateCost[1])/t)) #generate the
probability
    t=t-1
    if oldStateCost>=newStateCost and random() < p : #accept a worse
solution with a prob p!!
        state=new_state
    else:
        if oldStateCost<newStateCost:
            state=new_state

#print(f"The final state is: {state}")
#print(f"The sets are: {x.rows[state]}")
valid,cost = goal_check(state)
print(f"The solution is: {valid}. I reached the solution with {-cost}
sets, so I have a cost of {cost}")
print(f"Those are the covered sets: {set(np.concatenate(x.rows[state]))}")
```

```
The solution is: True. I reached the solution with 46 sets, so I have a
cost of -46
Those are the covered sets: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
```

Tabu search

Again the tweak is the same. The idea is to keep track of the states, so we can avoid to visit them again.

```
def tweak(state):
    new_state = state.copy()
    for i in range(int(30*num_points/100)): #change 30% of the points
        index = randint(0, num_points - 1)
        new_state[index] = not new_state[index]
    return new_state

def is_valid(state):
    v,c = goal_check(state)
    return v
```

```
state = initiale_state
tabu_list = []
for step in range(num_points):
    new_sol_list = [tweak(state) for _ in range(int(5*num_points/100))]
#generate a list of new solutions
    candidates = [sol for sol in new_sol_list if is_valid(sol) and sol not
in tabu_list] #if the solution is valid and not in the tabu_list
    if not candidates: #if no candidates, keep going
        continue
    for sol in candidates:          #take the best option from the
generated ones
        tabu_list.append(sol)
        if goal_check(sol)>goal_check(state):
            state = sol

#print(f"The final state is: {state}")
#print(f"The sets are: {x.rows[state]}")
valid,cost = goal_check(state)
print(f"The solution is: {valid}. I reached the solution with {-cost}
sets, so I have a cost of {cost}")
print(f"Those are the covered sets: {set(np.concatenate(x.rows[state]))}")
```

```
The solution is: True. I reached the solution with 23 sets, so I have a
cost of -23
Those are the covered sets: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
```

Iterated Local Search

The tweak is again the same as before. This is Hill Climbing but at the end we restart it.

```
def tweak(state):
    new_state = state.copy()
    for i in range(int(30*num_points/100)): #change 30% of the points
        index = randint(0, num_points - 1)
        new_state[index] = not new_state[index]
    return new_state
```

```
def hill_climbing(initiale_state):
    state = initiale_state
    for step in range(num_points):
        new_state = tweak(state)
        if goal_check(state)<goal_check(new_state):
            state=new_state
    return state
```

```
N_Restart = 10
state = initiale_state
for i in range(N_Restart):
    solution = hill_climbing(state)
    state = solution

valid,cost = goal_check(state)
print(f"The solution is: {valid}. I reached the solution with {-cost}
sets, so I have a cost of {cost}")
print(f"Those are the covered sets: {set(np.concatenate(x.rows[state]))}")
```

```
The solution is: True. I reached the solution with 22 sets, so I have a
cost of -22
Those are the covered sets: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85,
86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99}
```

Project -- Quixo

Quixo

```
import pickle
import math
import matplotlib.pyplot as plt
from random import randint, random, choice
from game import Move, Player
from CustomGameClass import Quixo as Game
from tqdm import trange
from typing import Literal
import numpy as np
import json
```

Reinforcement Learning Player

This class represents a player that uses Reinforcement Learning to make decisions in Quixo.

Attributes:

- `epochs` (int): The number of training epochs.
- `alpha` (float): The learning rate.
- `discount_factor` (float): The discount factor of the Bellman equation.
- `min_exploration_rate` (float): The minimum exploration rate during training.
- `exploration_decay_rate` (float): The rate at which the exploration rate decays during training.
- `opponent` (Player): The opponent player.
- `states` (list): A list to store the states visited during a game.
- `state_value` (dict): A dictionary to store the value of each state.
- `training_phase` (bool): A boolean to indicate if the player is training or not. It basically enables the exploration if it is true, otherwise it uses only the `state_value` dictionary to make decisions.

Methods:

- `give_rew(reward)`: Placeholder method for giving reward to the player.
- `add_state(state)`: Adds to the `states` array the state that a player has seen during a game.
- `reset()`: Reset the `states` array to be able to start a new game.
- `make_move(game)`: Chooses an action to take based on the current game state that can be random or based on the value of the dictionary. It takes from the dictionary, for each possible move, the value associated with the state of the board with the move performed. The maximum value will be the move to execute. We use the following recursive (bellman equation, temporal difference learning) formula to compute the state-value table: $$ V(S_t) \leftarrow V(S_t) + \alpha * (\gamma * V(S_t +1) - V(S_t)) $$ The formula simply tells us that the updated value of state t equals the current value of state t adding the difference between the value of next state and the value of current state, which is multiplied by a learning rate α.
- `update_state_value_table(reward)`: Updates the values of the `states_value` dictionary based on the states that the player has seen during the game and the reward that they have provided.
- `game_reward(player)`: Calculates the reward for the player in the current game.
- `train()`: Trains the player using reinforcement learning.
- `save_policy(name)`: Saves the state value table to a file.

- `load_policy(file)`: Loads the state value table from a file.

```
class RLPlayer(Player):
    def __init__(self, epochs: int,
                 alpha: float,
                 discount_factor: float,
                 min_exploration_rate: float,
                 exploration_decay_rate: float,
                 opponent: 'Player',
                 training_phase: bool) -> None:

        super().__init__()
        self.epochs = epochs
        self.alpha = alpha
        self.discount_factor = discount_factor
        self.exploration_rate = 1
        self.min_exploration_rate = min_exploration_rate
        self.exploration_decay_rate = exploration_decay_rate
        self.opponent = opponent
        self.training_phase = training_phase
        self.states=[]
        self.state_value = {}

    def give_rew(self, reward):
        pass

    def add_state(self, state):
        self.states.append(state)

    def reset(self):
        self.states = []

    def make_move(self, game: Game) -> tuple[tuple[int, int], Move]:
        available_moves = get_possible_moves(game,
game.get_current_player())
        if self.training_phase and (random() < self.exploration_rate):  #
do exploration
            return choice(available_moves)
        else:  # do exploitation
            value_max = -math.inf
            for move in available_moves:
                tmp = game.get_board()
                game._Game__move(move[0], move[1],
game.get_current_player())
                next_status =
convert_matrix_board_to_tuple(game.get_board())
                game.set_board(tmp)
                value = 0 if self.state_value.get(next_status) is None
else self.state_value.get(next_status)
                if value > value_max:
                    value_max = value
                    action = move
        return action
```

```
    def update_state_value_table(self, reward):
        for st in reversed(self.states):
            if self.state_value.get(st) is None:
                self.state_value[st] = 0
            current_q_value = self.state_value[st]
            reward = current_q_value + self.alpha * (self.discount_factor
* reward - current_q_value)
            self.state_value[st] = reward

    def game_reward(self, player: 'RLPlayer')-> Literal[-10, 10]: #Literl
is a type that indicates that is literally accepts only -10 and 10 as
values.
        if self == player:
            return 10
        else:
            return -10

    def train(self, player_name='') -> None:
        game = Game()
        all_rewards = []
        # define how many episodes to run
        pbar = trange(self.epochs) #same thing as range, but with tqdm
(visual thigngs...)
        # define the players
        players = (self, self.opponent)

        for epochs in pbar:
            rewards = 0
            winner = -1
            players = (players[1], players[0])
            player_idx = 1

            while winner < 0:
                # change player
                player_idx = (player_idx + 1) % 2
                player = players[player_idx]
                game.switch_player()

                ok = False
                if self == player:
                    while not ok:
                        from_pos, slide = self.make_move(game)
                        ok = game._Game__move(from_pos, slide,
game.get_current_player())
                        state_after_move =
convert_matrix_board_to_tuple(game.get_board())
                        self.add_state(state_after_move)

                else:
                    while not ok:
                        from_pos, slide = player.make_move(game)
                        ok = game._Game__move(from_pos, slide,
game.get_current_player())
```

```
                winner = game.check_winner()

            # update the exploration rate
            #np.clip(a,min,max), clips the value between min and max
            self.exploration_rate = np.clip(
                np.exp(-self.exploration_decay_rate * epochs),
self.min_exploration_rate, 1
            )

            reward = self.game_reward(player)
            self.update_state_value_table(reward)
            rewards += reward
            self.reset()
            game.reset()
            all_rewards.append(rewards)
            pbar.set_description(f'rewards value: {rewards}, current
exploration rate: {self.exploration_rate:2f}')

        plot_training_trends(all_rewards, filename=f"{player_name} trained
against {self.opponent.__class__.__name__}")

        print(f'** Last 50_000 episodes - Mean rewards value:
{sum(all_rewards[-50_000:]) / 50_000:.2f} **')

    def save_policy(self, name):
        fw = open(name, 'wb')
        pickle.dump(self.state_value, fw, protocol=4)
        fw.close()

    def load_policy(self, file):
        fr = open(file, 'rb')
        self.state_value = pickle.load(fr)
        fr.close()
```

MinMax Player

This class represents a player who uses the MinMax algorithm to make decisions in the game. The MinMax algorithm is a search algorithm that is used in two-player games to make optimal decisions.

Attributes:

- `playerPlaying` (int): The player who is currently playing.
- `levels_depth` (list): A list of tuples, where each tuple contains the depth level and the maximum number of possible moves for that depth level.

Methods:

- `game_evaluation`: Evaluate the game state based on the current player and depth. If the game has ended and the winner is the one that is playing, the reward is 100 + the actual depth, otherwhise is -100 - the actual depth.

- `min_max`: Perform the Minimax algorithm to determine the best move for the current player. This method takes as input the current game, the alpha and beta values (used for alpha-beta pruning, a technique for reducing the number of nodes evaluated by the MinMax algorithm), and the current depth of the search tree. If the depth is zero or if there is a winner in the game, the method returns the game rating and no moves. Otherwise, for each possible move, it creates a copy of the game, makes the move, and recursively calls the min_max method on the copy of the game. If the returned rating is greater than alpha, alpha is updated and the move is considered the best move. If beta is less than or equal to alpha, the cycle stops for alpha-beta pruning. The process is similar for the opponent, with the difference that we try to minimize the rating instead of maximizing it.
- `choose_action`: Choose the best action (move) for the current player using the Minimax algorithm. The method starts by getting all possible moves for the current player using the `get_possible_moves` function. It then calculates the search depth for the MinMax algorithm based on the number of possible moves. This is done through a for loop that passes through the `levels_Depth` list. If the number of possible moves is greater than a certain value, the depth is set to a certain level. We do this because if we have too many possible moves we will have lower depth otherwise the search would be infinite... For example if we have this [(4,0),(3,23),(2,28),(1,32)], and we have 10 moves we will look at 4 level of depth, but if we have many more possible moves, such as 30, we will look just at the 2 depth. The cycle stops as soon as a level is found that does not exceed the number of possible moves. The method then calls the `min_max` method to determine the best move for the current player. Finally, the method returns the best move.

```
class MinMaxPlayer(Player):
    def __init__(self, playerPlaying, levels_depth):
        super().__init__()
        self.moves_value = []
        self.playerPlaying = playerPlaying
        self.levels_depth = levels_depth

    def game_evaluation(self, game: Game, depth):
        win = game.check_winner()
        ret = 0 + depth
        if win == 0 and self.playerPlaying == 0:
            ret = 100 + depth
        elif win == 0 and self.playerPlaying == 1:
            ret = -100 - depth
        elif win == 1 and self.playerPlaying == 1:
            ret = 100 + depth
        elif win == 1 and self.playerPlaying == 0:
            ret = - 100 - depth
        return ret

    def min_max(self, game: 'Game', alpha, beta, depth):
        if depth <= 0 or game.check_winner() != -1:
            return self.game_evaluation(game, depth), None
        best_move = None
        if game.current_player == self.playerPlaying:
            for move in get_possible_moves(game,
game.get_current_player()):
                tmp = game.get_board()
                g = Game()
```

```
                g.set_board(tmp)
                g.current_player = self.playerPlaying
                g._Game__move(move[0], move[1], g.get_current_player())
                g.current_player = 1 - self.playerPlaying
                eval, _ = self.min_max(g, alpha, beta, depth - 1)
                if eval > alpha:
                    alpha = eval
                    best_move = move
                if beta <= alpha:
                    break
            return alpha, best_move
        else:
            for move in get_possible_moves(game,
game.get_current_player()):
                tmp = game.get_board()
                g = Game()
                g.set_board(tmp)
                g.current_player = 1 - self.playerPlaying
                g._Game__move(move[0], move[1], g.get_current_player())
                g.current_player = self.playerPlaying
                eval, _ = self.min_max(g, alpha, beta, depth - 1)
                if eval < beta:
                    beta = eval
                    best_move = move
                if beta <= alpha:
                    break
            return beta, best_move

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        possibleMoves = get_possible_moves(game,
game.get_current_player())
        possibleMoveCount = len(possibleMoves)

        depth = 0
        for depthLvl in self.levels_depth:
            if possibleMoveCount > depthLvl[1]:
                depth = depthLvl[0]
            else:
                break

        _, move = self.min_max(game, -math.inf, math.inf, depth)

        return move

    def give_rew(self, reward):
        pass
```

Utility functions

- `convert_matrix_to_tuple`: Convert the matrix board representation to a tuple representation.

- `get_possible_moves`: Returns a list of possible moves for the player. It returns a list of tuples, where each tuple contains the move coordinates and the move direction. Each move is a tuple of the form (row, column, direction). The direction is an enum Move that can be either 'up', 'down', 'left' or 'right'.
- `test_player`: Test the performance of a player against another player. The method takes as input the two players, the number of games to play, and the name of the two players just to specify them in the plot. The method prints the number of wins for the first player and the number of wins for the second player.
- `plot_total_win_rate`: Creates a bar graph to display the total number of wins of two players in a game. The generated image is saved in the `images` folder.
- `plot_training_trends`: Creates a graph to visualize the trend of total rewards while training a reinforcement learning agent. Within the function, the average of the rewards is calculated every 500 training steps. The generated image is saved in the `images` folder.

```
def convert_matrix_board_to_tuple(board):
    """
    Converts a matrix board to a tuple representation.

    Args:
        board (list): The matrix board to be converted.

    Returns:
        tuple: The converted tuple representation of the board.
    """
    current_board = tuple(tuple(riga) for riga in board)
    return current_board

def get_possible_moves(game: 'Game', player: int) -> list[tuple[tuple[int,
int], Move]]:
    """
    Get a list of possible moves for a given player in the game.

    Args:
        game (Game): The game object representing the current state of the
game.
        player (int): The player for whom to find the possible moves.

    Returns:
        list[tuple[tuple[int, int], Move]]: A list of tuples, where each
tuple contains the coordinates of a possible move
        and the corresponding move direction.

    """
    # possible moves:
    # - take border empty and fill the hole by moving in the 3 directions
    # - take one of your blocks on the border and fill the hole by moving
in the 3 directions
    # 44 at start possible moves
    pos = set()
    for r in [0, 4]:
        for c in range(5):
```

```
            if game.get_board()[r, c] == -1 or game.get_board()[r, c] ==
player:
                if r == 0 and c == 0:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 0 and c == 4:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.LEFT))
                elif r == 4 and c == 0:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 4 and c == 4:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                elif r == 0:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.LEFT))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 4:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                    pos.add(((c, r), Move.RIGHT))
    for c in [0, 4]:
        for r in range(5):
            if game.get_board()[r, c] == -1 or game.get_board()[r, c] ==
player:
                if r == 0 and c == 0:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 0 and c == 4:  # OK
                    pos.add(((c, r), Move.BOTTOM))
                    pos.add(((c, r), Move.LEFT))
                elif r == 4 and c == 0:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.RIGHT))
                elif r == 4 and c == 4:  # OK
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                elif c == 0:
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.RIGHT))
                    pos.add(((c, r), Move.BOTTOM))
                elif c == 4:
                    pos.add(((c, r), Move.TOP))
                    pos.add(((c, r), Move.LEFT))
                    pos.add(((c, r), Move.BOTTOM))
    return list(pos)

def test_player(player1, player2, num_games, name_player1, name_player2):
    """
    Test the performance of two players in a series of games.

    Parameters:
    player1 (object): The first player object.
```

```
    player2 (object): The second player object.
    num_games (int): The number of games to be played.
    name_player1 (str): The name of the first player.
    name_player2 (str): The name of the second player.

    Returns:
    None
    """
    g = Game()
    player1_wins = 0
    player2_wins = 0
    draws = 0
    games = 0
    for _ in range(num_games):
        winner = g.play(player1, player2)
        games += 1
        g.reset()
        if winner == 0:
            player1_wins += 1
        if winner == 1:
            player2_wins += 1
        if winner == -1:
            draws += 1

    plot_total_win_rate(player1_wins, player2_wins, draws, name_player1,
name_player2)
    print(f"{name_player1} won {player1_wins / num_games * 100}%")
    print(f"{name_player2} won {player2_wins / num_games * 100}%")
    print(f"Draws: {draws / num_games * 100}%")

def plot_total_win_rate(wins_player1, wins_player2, draws, name_player1,
name_player2):
    """
    Plots the total win rate of two players.

    Parameters:
    - wins_player1 (int): Number of wins for player 1.
    - wins_player2 (int): Number of wins for player 2.
    - draws (int): Number of draws.
    - name_player1 (str): Name of player 1.
    - name_player2 (str): Name of player 2.

    Returns:
    None
    """
    plt.bar([name_player1, name_player2, 'draws'], [wins_player1,
wins_player2, draws], color=['blue', 'orange', 'green'])
    plt.ylabel('Number of games')
    plt.savefig(f"images/{name_player1} wins vs {name_player2} wins.png")
    plt.show()

def plot_training_trends(total_rewards: [int], filename=''):
    """
    Plots the training trends of the total rewards.
```

```
    Args:
        total_rewards (list[int]): List of total rewards obtained during
training.
        filename (str, optional): Name of the file to save the plot.
Defaults to ''.
    """
    mean_array = np.mean(np.array(total_rewards).reshape(-1, 500), axis=1)
    index = np.arange(0, len(total_rewards), 500)
    plt.plot(index, mean_array, label='Mean rewards value')
    plt.ylabel('Mean rewards value')
    plt.xlabel('Epochs')
    plt.savefig(f"images/{filename} training_trends.png")
    plt.show()
```

Random Player Definition

```
class RandomPlayer(Player):
    def __init__(self) -> None:
        super().__init__()

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
        from_pos = (randint(0, 4), randint(0, 4))
        move = choice([Move.TOP, Move.BOTTOM, Move.LEFT, Move.RIGHT])
        return from_pos, move

    def give_rew(self, reward):
        pass

    def add_state(self, s):
        pass
```

Human Player Definition

```
class HumanPlayer(Player):
    def __init__(self) -> None:
        super().__init__()

    def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
            available_moves = get_possible_moves(game,
game.get_current_player())
            while True:
                row = int(input("Input your action row:"))
                col = int(input("Input your action column:"))
                from_pos = (col, row)
                move = int(input("Input your action move: (1 for top, 2
for bottom, 3 for left, 4 for right):"))
```

```
                if move == 1:
                    move = Move.TOP
                elif move == 2:
                    move = Move.BOTTOM
                elif move == 3:
                    move = Move.LEFT
                elif move == 4:
                    move = Move.RIGHT
                else:
                    print("Invalid move, please input again")
                    continue

                if (from_pos, move) in available_moves:
                    return from_pos, move
                else:
                    print("Invalid move, please input again")
                    continue

    def give_rew(self, reward):
        pass

    def add_state(self, s):
        pass
```

Hyperparameters

- `epochs`: training epochs
- `alpha`: learning rate
- `discount_factor`: the discount rate of the Bellman equation
- `min_exploration_rate`: the minimum rate for exploration during the training phase
- `exploration_decay_rate`: the exploration decay rate used during the training
- `training_phase`: a boolean value that indicates if the player is in training phase or not. It basically enables the exploration if it is true, otherwise it uses only the `state_value` dictionary to make decisions.
- `RandomP`: the opponent to play against that use a Random strategy for the RL training
- `MinMaxP`: the opponent to play against that use a MinMax strategy for the RL testing. It takes as input the `level_depth`, a list of tuples, where each tuple contains the depth level and the maximum number of possible moves for that depth level. For example if there are 5 possible moves, the depth level is 4, if there are 40 possible moves, the depth level is 1.
- `num_games`: number of games for testing

```
epochs = 850000
alpha = 0.1
discount_factor = 0.95
min_exploration_rate=0.01
exploration_decay_rate=5e-6
training_phase=True
RandomP = RandomPlayer()
```

```
MinMaxP = MinMaxPlayer(0, [(4,0),(3,23),(2,28),(1,32)])
num_games = 1000
```

Let's do some computation: RL Player trained against Random Player

```
# create the RL player
rl_agent_RandomOpponent = RLPlayer(
    epochs=epochs,
    alpha=alpha,
    discount_factor=discount_factor,
    min_exploration_rate=min_exploration_rate,
    exploration_decay_rate=exploration_decay_rate,
    opponent=RandomP,
    training_phase=training_phase
)
# train the RL player
rl_agent_RandomOpponent.train(player_name='RL_Player_1')
```

Test Reinforcement Learning Player vs Random Player

The `RL_player_1` is the best player with this parameters:

- `epochs` = 500_000,
- `alpha` = 0.1,
- `discount_factor` = 0.95,
- `min_exploration_rate` = 0.01,
- `exploration_decay_rate` = 1e-5, so you can get it by training with this parameters, and save it with the method `save_policy()`

```
rl_player = RLPlayer(
    epochs=epochs,
    alpha=alpha,
    discount_factor=discount_factor,
    min_exploration_rate=min_exploration_rate,
    exploration_decay_rate=exploration_decay_rate,
    opponent=RandomP,
    training_phase=False
)

rl_player.load_policy('RL_player_1')
```

```
test_player(rl_player, RandomP, num_games, 'RL_player_1(first_move)',
'Random Player')
test_player(RandomP, rl_player, num_games, 'Random Player',
'RL_player_1(second_move)')
```

RL Player results

- RL Player 1
    - `epochs` = 500_000,
    - `alpha` = 0.1,
    - `discount_factor` = 0.95,
    - `min_exploration_rate` = 0.01,
    - `exploration_decay_rate` = 1e-5,

    Results

    - length of `states_value` dictionary: 3_988_351
    - Last 50000 episodes - Mean rewards value: 6.44
    - win rate vs `RandomPlayer` in 1000 games (RL always first move) - 92%
    - win rate vs `RandomPlayer` in 1000 games (Random always first move) - 78%
- RL Player 2
    - `epochs` = 750_000,
    - `alpha` = 0.1,
    - `discount_factor` = 0.95,
    - `min_exploration_rate` = 0.01,
    - `exploration_decay_rate` = 5e-6,

    Results

    - length of `states_value` dictionary: 6_610_522
    - Last 50000 episodes - Mean rewards value: 5.78
    - win rate vs `RandomPlayer` in 1000 games (RL always first move) - 88%
    - win rate vs `RandomPlayer` in 1000 games (Random always first move) - 63%
- RL Player 3
    - `epochs` = 850_000,
    - `alpha` = 0.1,
    - `discount_factor` = 0.95,
    - `min_exploration_rate` = 0.01,
    - `exploration_decay_rate` = 5e-6,

    Results

    - length of `states_value` dictionary: 8_727_589
    - Last 50000 episodes - Mean rewards value: 5.39
    - win rate vs `RandomPlayer` in 1000 games (RL always first move) - 83%
    - win rate vs `RandomPlayer` in 1000 games (Random always first move) - 71%

RL Player 1 is the best policy we obtained, and we saved it in `RL_player_1`. We will use it for the next tests.

Test MinMax Player vs Random Player

```
test_player(MinMaxP, RandomP, num_games, 'MinMax Player(first_move)',
'Random Player')
test_player(RandomP, MinMaxP, num_games, 'Random Player', 'MinMax
Player(second_move)')
```

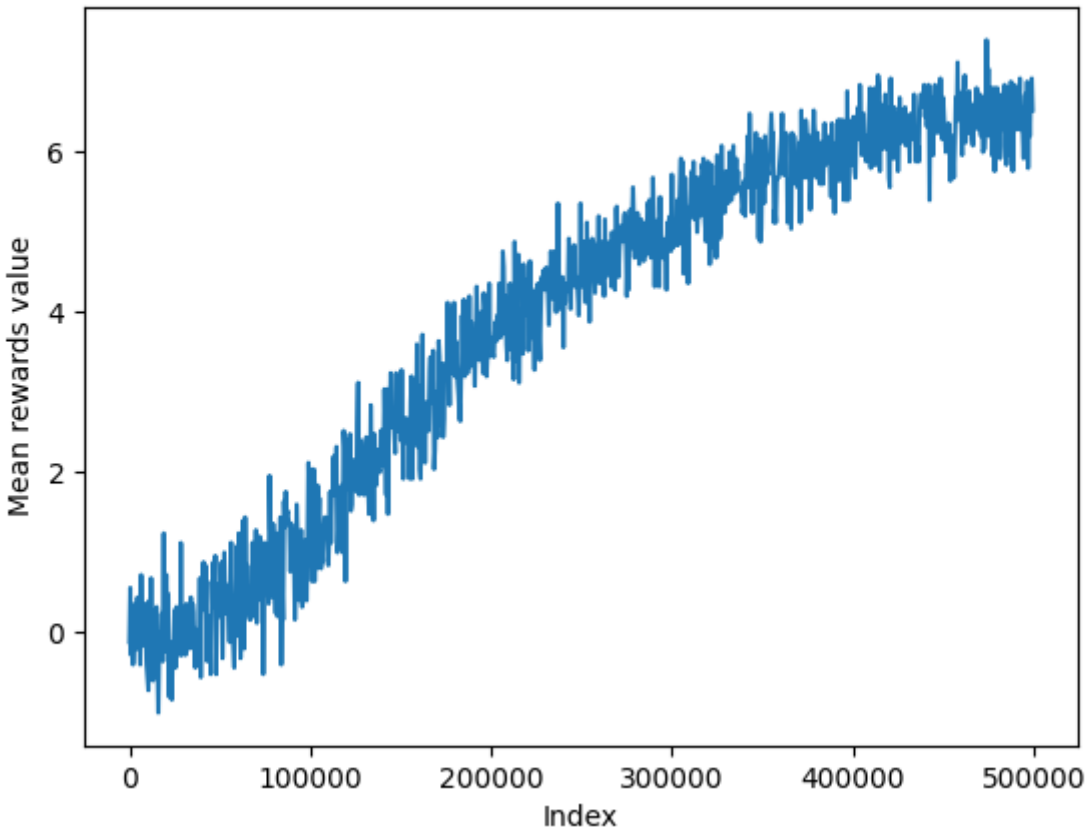
Results and conclusions

TODO commentare i risultati e ordinarli un po meglio

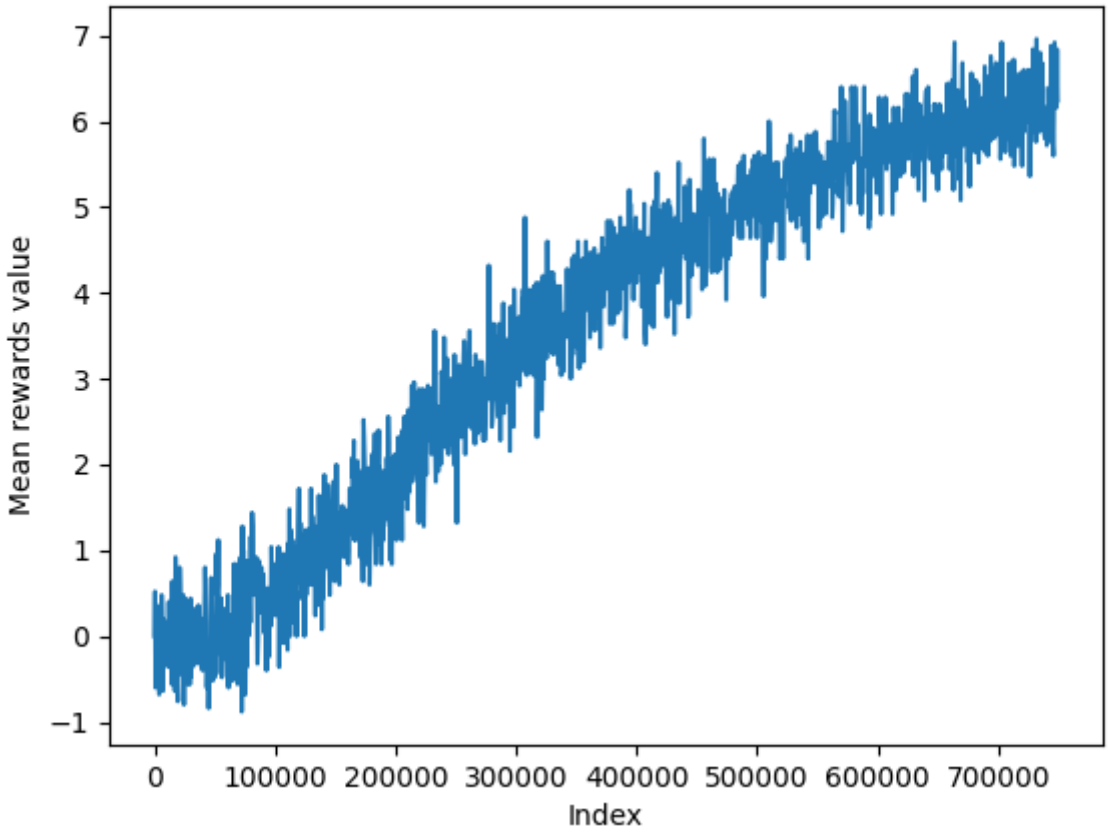
We consider a game as a draw if it lasts more than 200 moves (100 moves for player). This is because the game can last forever. In fact, if the two players play optimally, the game will never end.

Reinforcement Learning training trends

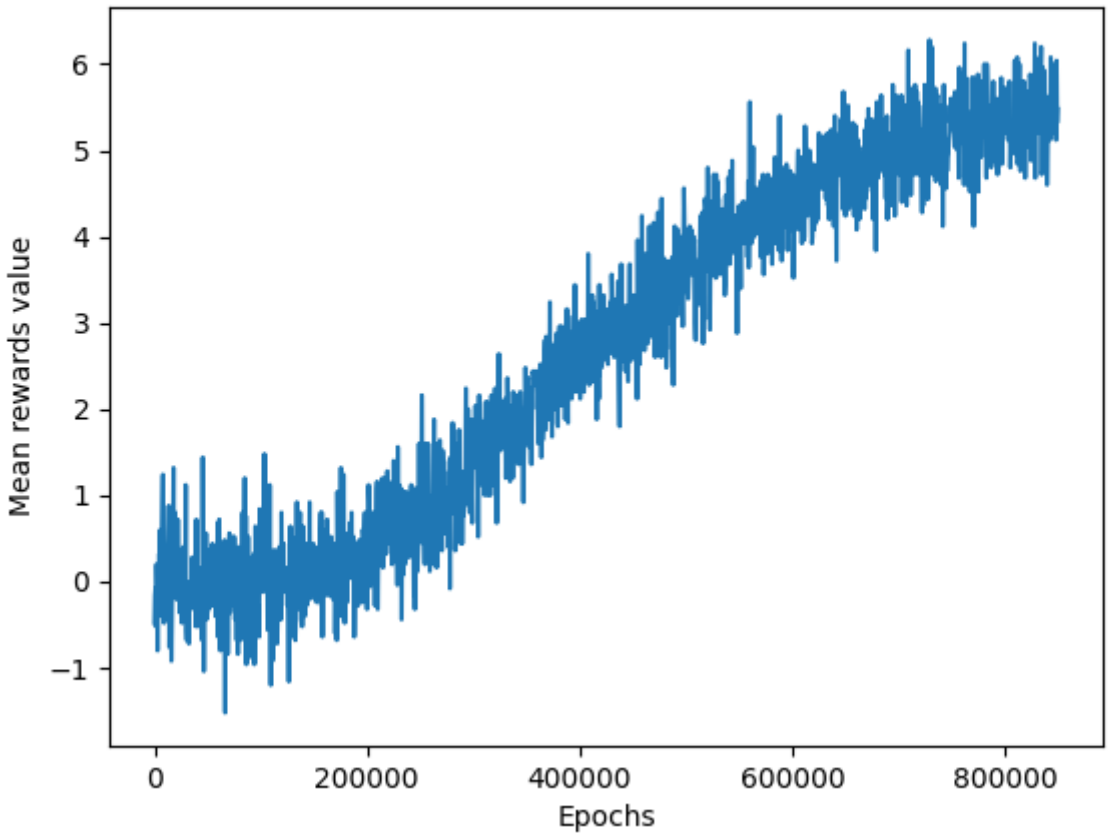
RL Player 1



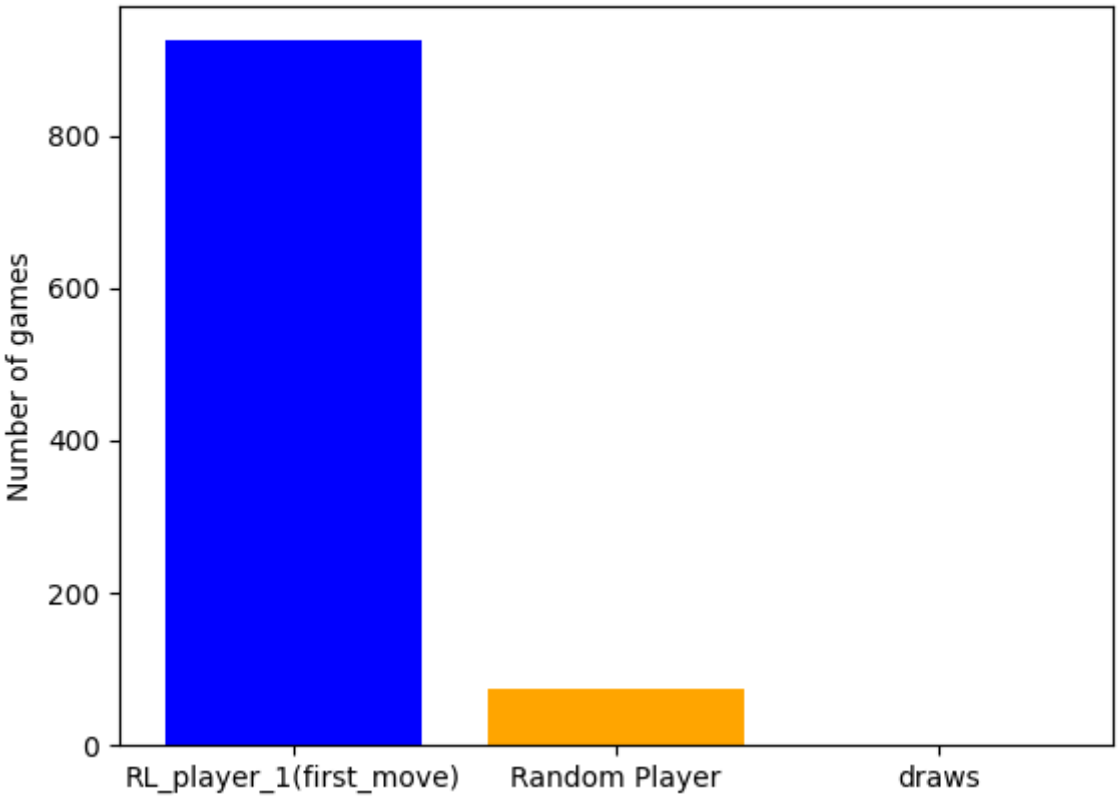
RL Player 2

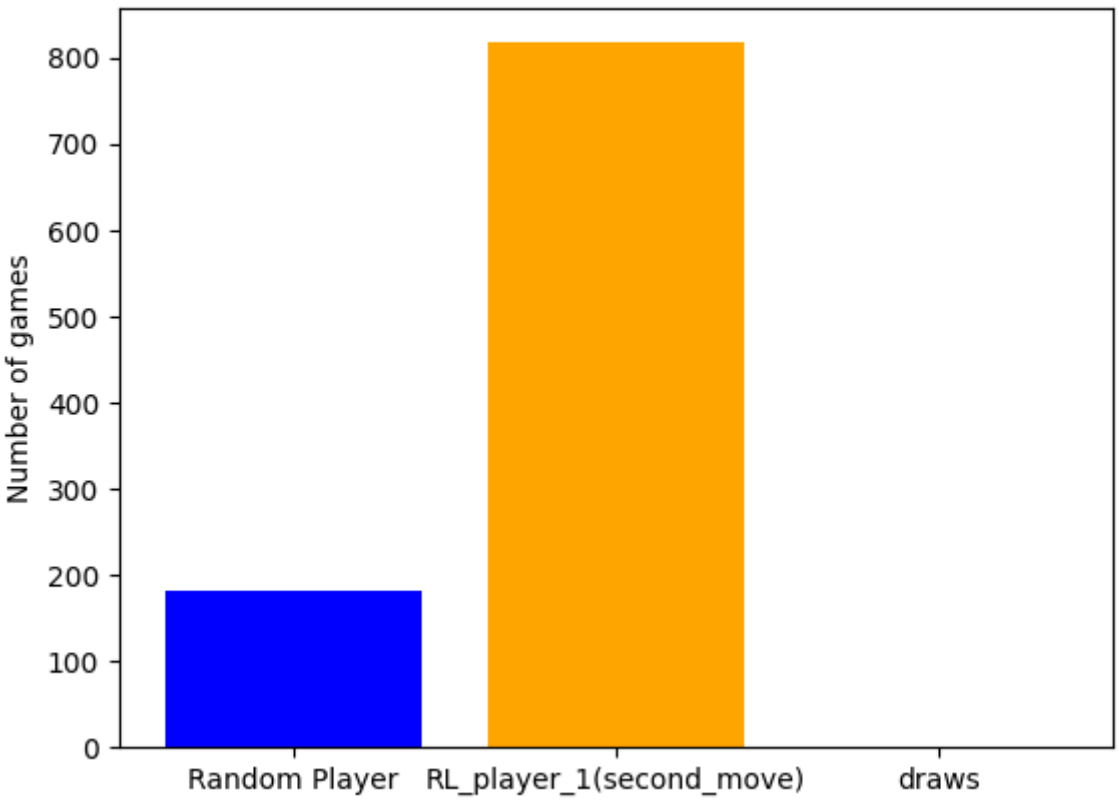

Mean rewards value
Index
7
6
5
4
3
2
1
0
−1
0
100000
200000
300000
400000
500000
600000
700000

RL Player 3

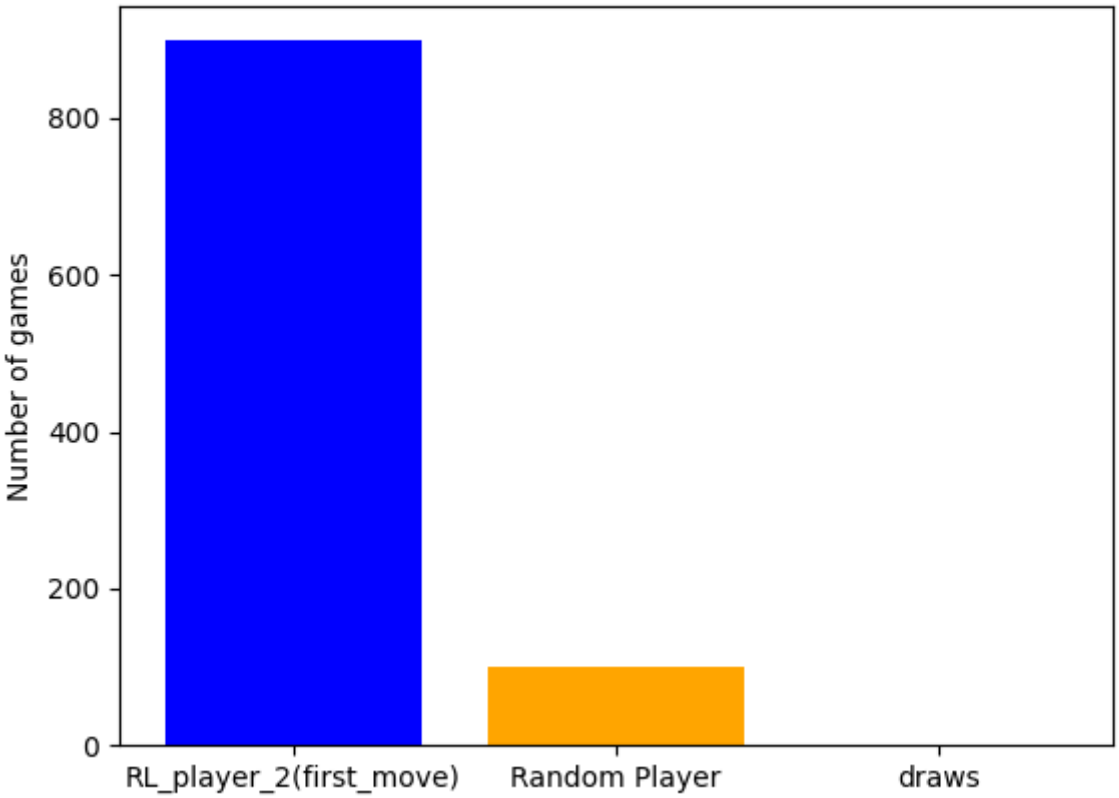

Mean rewards value
Epochs
6
5
4
3
2
1
0
−1
0
200000
400000
600000
800000

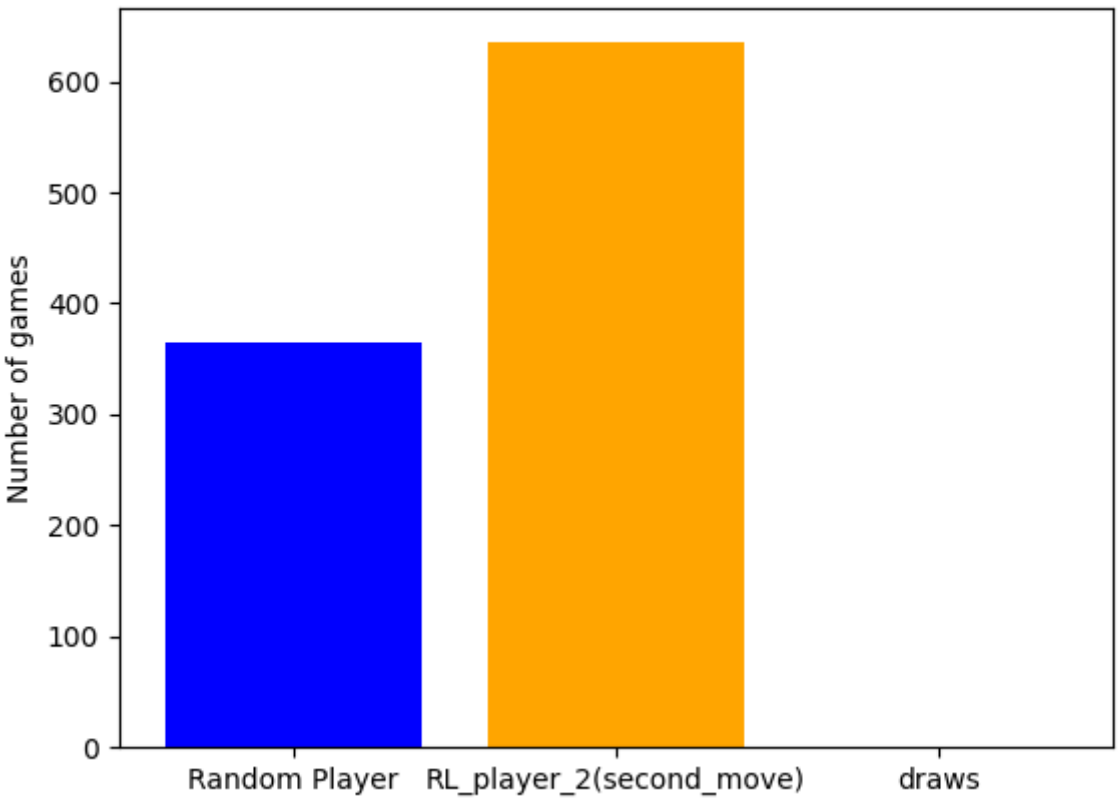
Reinforcement Learning (RL_player_1) vs Random Player



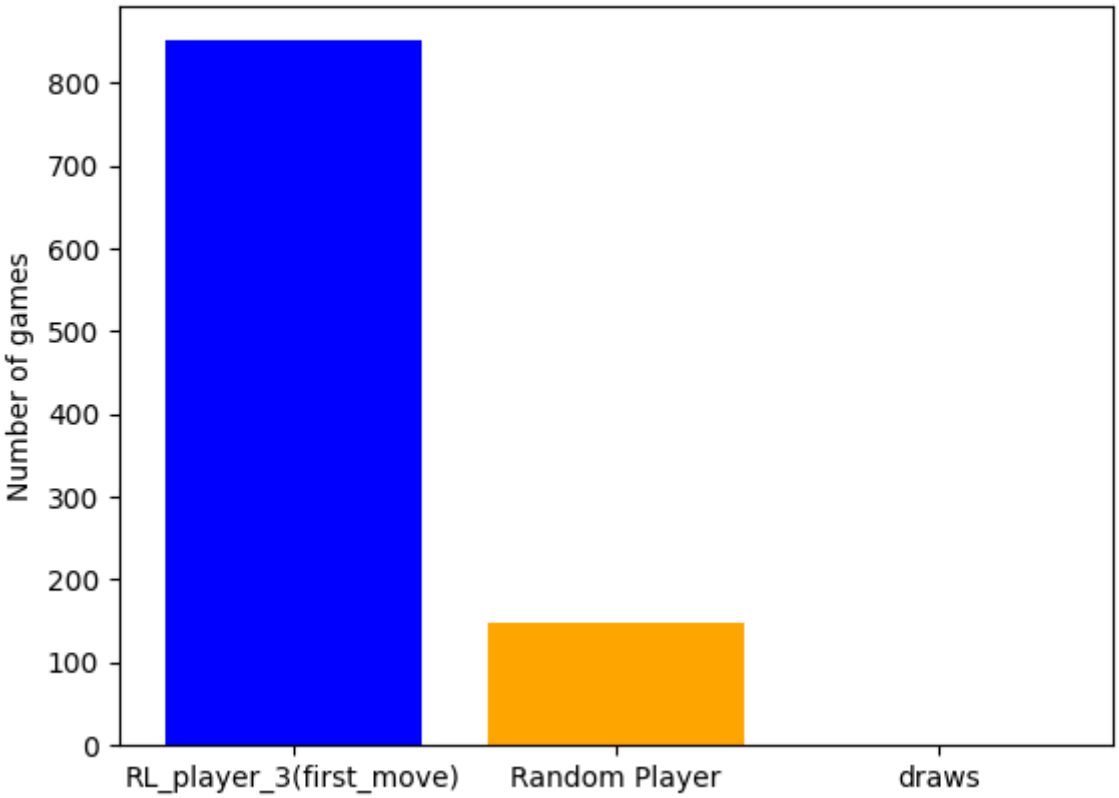


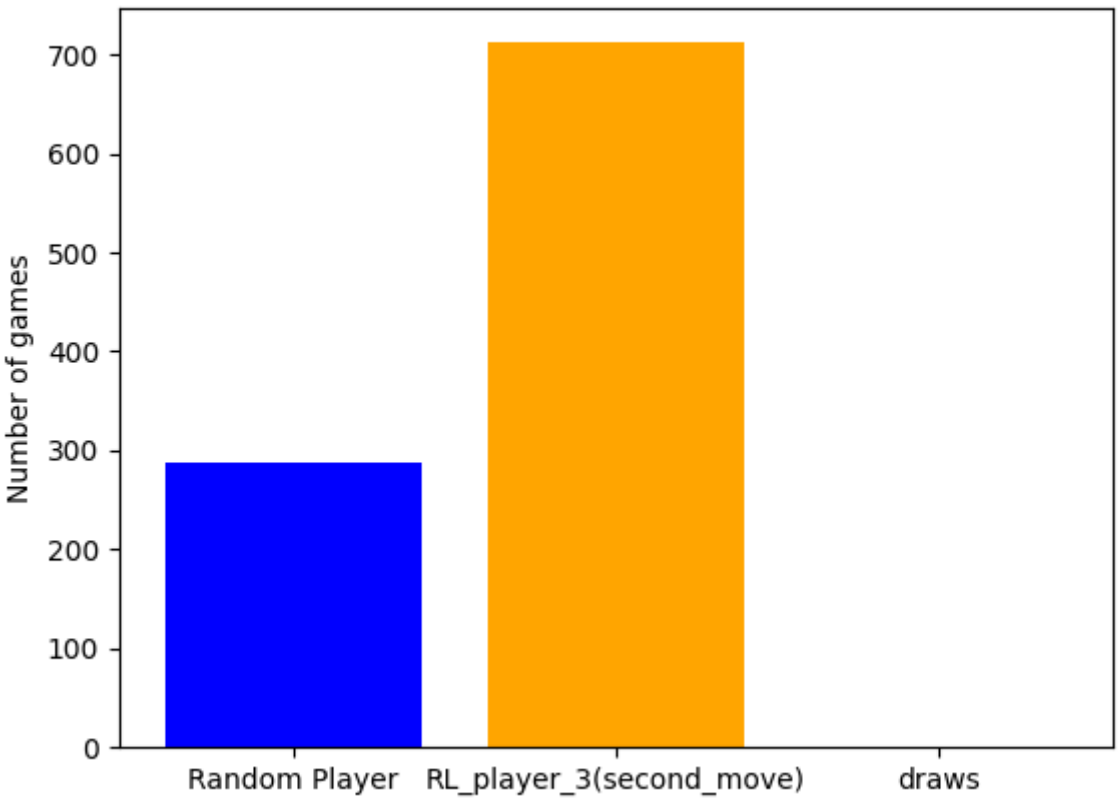
Reinforcement Learning (RL_player_2) vs Random Player



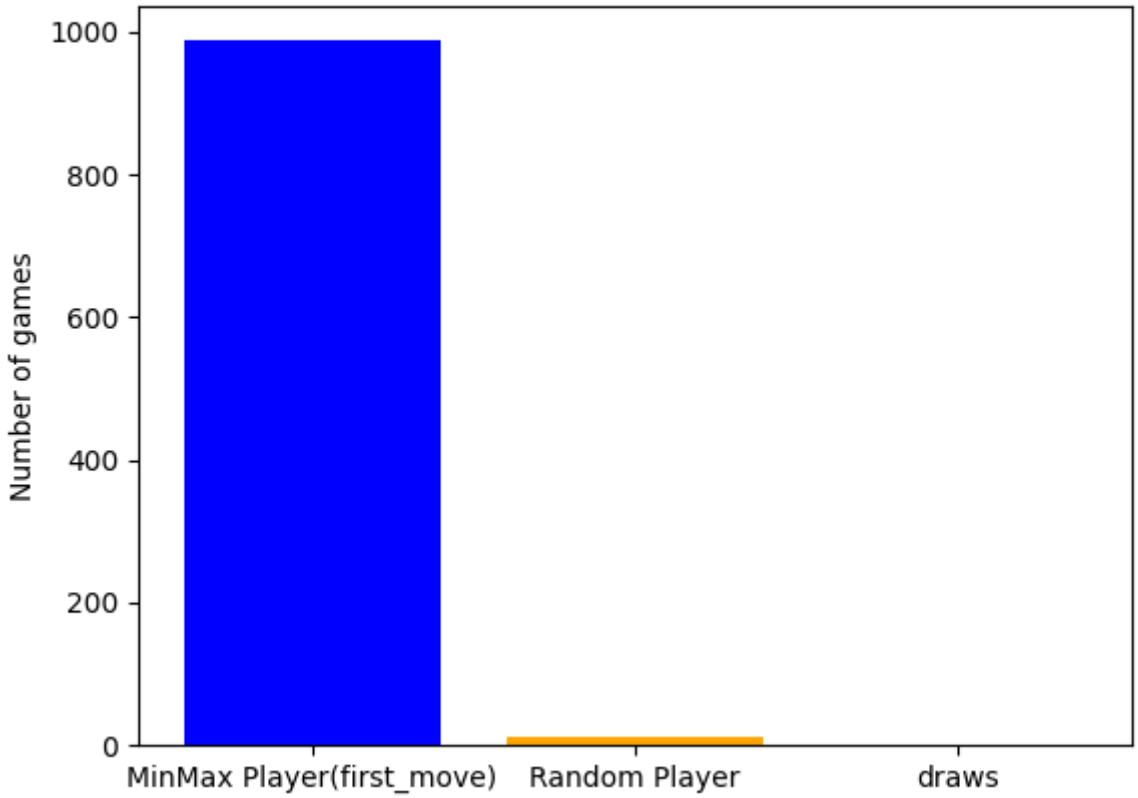


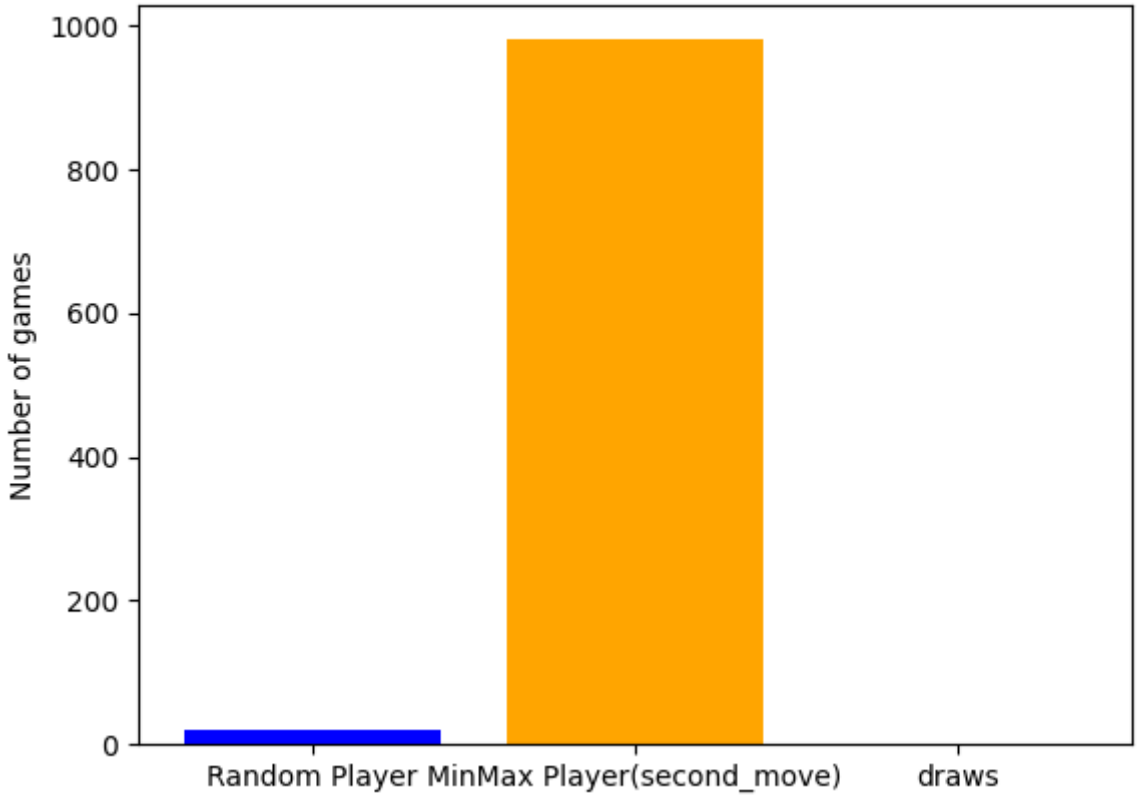
Reinforcement Learning (RL_player_3) vs Random Player





MinMax Player vs Random Player





Let's play!

```
''' board indexes

0
1
2
3
4
  0 1 2 3 4

'''

rl_player = RLPlayer(
    epochs=epochs,
    alpha=alpha,
    discount_factor=discount_factor,
    min_exploration_rate=min_exploration_rate,
    exploration_decay_rate=exploration_decay_rate,
    opponent=RandomP,
    training_phase=False
)
human_player = HumanPlayer()

rl_player.load_policy('RL_player_1')
g = Game()
winner = g.play(rl_player, human_player, print_flag=True)
```