FINAL LOG OF COMPUTATIONAL INTELLIGENCE ACTIVITY

Claudio Macaluso - s317149

LAB01 - A*

Intro

In the lab01, it was required to solve the set covering problem using the A* method, creating a new heuristic in addition to the already provided one (distance-based heuristic)

Brief description of the problem

The goal of the set covering problem is to find the smallest collection of sets from a given family of sets such that each element from the universal set is covered by at least one of the selected sets.
Let x_i be a binary decision variable that indicates wheather set s_i is selected.
The objective is to minimize the total cost:

$$\sum_i c_i \cdot x_i$$

where c_i is the cost for selecting x_i.
The whole formulation is subject to the constraint that each element in U (the universal set) is covered by at least one selected set.

Brief description of the method

A* algorithm is a pathfinding algorithm that takes as evaluation function, i.e. a function that describes the priority-goodness of an action, the result of:

$$f(n) = g(n) + h(n)$$

- $g(n)$: the cost of the path from the start node to node n
- $h(n)$: heuristic -> estimate of the cost from the node n to the goal node

A good heuristic must comply some constraints:

- Admissibility: it never overestimates the true cost to reach the goal, i.e. $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost
- Consistency (or monotonicity): for every state n and its successor m we have: $h(n) \leq g(n, m) + h(m)$, where $g(n, m)$ is the cost of the edge from n to m

Implementation details

With the heuristic created, I tried to find a way to make the resulting cost more pessimistic, by calculating more precisely how many sets do we have to take to bring our agent to the goal.

```
def h01(state):
  already_covered = covered(state)
  if np.all(already_covered):
    return 0
  missing_size = PROBLEM_SIZE - sum(already_covered)
  candidates = sorted((
      sum(np.logical_and(s, np.logical_not(already_covered))) for s in SETS),
      reverse=True
    )
  taken = 1
  while sum(candidates[:taken]) < missing_size:
    taken += 1
  return taken

# try to recursively estimate the nuumber of sets we have to take to cover all
def h02(state, j=0):
  # get the already covered elements with state
  already_covered = covered(state)
  # if all elements are covered, expected cost is zero
  if np.all(already_covered):
    return 0
  candidates = []
  for i in range(len(SETS)):
    # append (number of elements that would be covered (discarding already covered), index
    candidates.append((
        sum(np.logical_and(SETS[i], np.logical_not(already_covered))),
        i
      ))
  # take the best fitting set from the untaken
  candidate = max(candidates)[1]
  tmp_state = State(
      state.taken ^ {candidate},
      state.not_taken ^ {candidate},
    )
  # decrement the number of levels you want to explore
  if j>0:
    j-=1
  # if we reach the bottom, use the old distance based heuristic
  if j==0:
    taken = 1 + h01(tmp_state)
  # else explore another level (further decision)
  else:
    taken = 1 + h02(tmp_state, j)
  return taken
```

Conclusions

Sadly, the new heuristic is suboptimal due to its non admissibility, because it can overestimate the cost from a state to the goal: this is caused by the fact that it takes the best matching as first set, but it could possibly happend that a worse matching set can lead to a minor cost.
Besides the suboptimality of the heuristic, it seems to perform pretty well in most of the cases.

LAB02 - Nim ES

Intro

In the delivery was asked to try to implement an Evolutive Strategy able to play Nim.

Brief description of the problem

Nim is a two-player game where the players take turns removing a number $1 \leq n \leq k$ of objects from a set of heaps. The game starts with a number of heaps, each containing a variable number of objects. The player who remove the last object or objects lose (misère variant).

Brief description of the method

Evolutive strategy algorithm are a family of population-based optimization algorithm that leverage different genetic operators (i.e. crossover and mutation) to generate new solution from the previous ones. Through these new solutions, a selection is performed based on the result of a so called "fitness function": it is used to assess the quality of the solution itself. The whole process is repeated a number of time to obtain the optimal solution.

Implementation details

Evolutive strategies implemented in this lab works in this way:

- Initialize a number of parameters (in this case just one)
- Evaluate the quality of the current value of the parameters
- Generate λ new solution from the current one using genetic operators (in this case just mutation)
- Take the new solution between the newly generated solutions in a specific way given the type of algorithm we are implementing
- Repeat the process

The mutation used is Gaussian mutation, that takes parameters and tweak them sampling from a Gaussian distribution with $\mu =$ parameter values and $\sigma = 0.1$. This was used to generate offspring (new solutions) from the current one:

```
def generate_offspring(self) -> list:
    offspring = []
    for _ in range(self.l):
      # self.params = current parameters, self.s = choosen sigma
      params = np.random.normal(loc=self.params, scale=self.s)
      for i, _ in enumerate(params):
        # used to normalize to an interval [0,1] (probability value)
        while params[i] > 1 or params[i] < 0:
          params[i] = np.random.normal(loc=params[i], scale=self.s)
      offspring.append(params)
    return offspring
```

The proposed solution consists in three different algorithm types:

- $(1 + 1)$: take the current solution and generate a new one; if the new solution is better, set it as the current one
- $(1 + \lambda)$: take the current solution and generate λ new solutions; if there's a better solution inside the λ new ones, set it as the current one
- $(1, \lambda)$: take the current solution and generate λ new solutions; discard the current one and set the maximum from the λ new solutions as the current one

This optimized parameters are then used by the program to actually play and even to evaluate the solution, performing a number of games and returning the win rate of the strategy given such parameters.

Leave-one strategy

The strategy that I've implemented consists in taking the parameter as the percentage of heaps to leave with just one object inside (we'll call it jo heap). What the strategy do is:

- Begin taking from the heaps $n-1$ elements, being n the number of elements in the heap
- After that the desired number of jo heaps are present, start taking all the elements from the others non jo heaps
- If the remaining element from one of the jo heaps is taken from the opponent, generate another jo heap (if possible) as described before
- Do this until the game has finished

The strategy is implemented as follow:

```
def leave_one_strategy(params:list, state: Nim) -> Nimply:
  # take the number of jo heaps to leave
  needed = round(params[0] * len(state.rows))

  # take the number of jo heaps already present
  ones = sum([1 for _ in state.rows if _ == 1])
  # take the number of heaps with number of elements greater than one
```

```
gt_ones = sum([1 for _ in state.rows if _ > 1])

# take randomly a heap with a number of elements >= 1
r = 0
while True:
  r = round(np.random.rand() * (len(state.rows)-1))
  if (state.rows[r] >= 1):
    break

# IF
# you have less jo heaps than needed (have to create them!) &
# there are some heaps with number of elements greater than one &
# you have enough number of non-zero heaps
if ones < needed and gt_ones != 0 and needed <= (ones+gt_ones):
  # generate ones until it's possible => take randomly a heap with a number of elements >
  while(state.rows[r] <= 1):
    r = round(np.random.rand() * (len(state.rows)-1))
  return Nimply(r, state.rows[r]-1)
# ELIF
# you have the right number-excess of jo heaps &
# there are some heaps with number of elements greater than one &
# you have enough number of non-zero heaps
elif ones >= needed and gt_ones != 0 and needed <= (ones+gt_ones):
  # set to zero gt_ones heaps
  while(state.rows[r] <= 1):
    r = round(np.random.rand() * (len(state.rows)-1))
  return Nimply(r, state.rows[r])
else:
  # set to zero all heaps
  return Nimply(r, state.rows[r])
```

Conclusions

Using the proposed solution with `ITERATIONS = 100` and a heaps number `R=20`, the optimal strategy found is to leave `7/20` jo heaps before removing all the remaining heaps. This lead to an average of `96%` of wins against a random strategy.

Peer review

For this lab, I wrote a review on the code of Angelo Iannelli - https://github.com/AngeloIannielli/polito-computational-intelligence-23

LAB09

Intro

In lab09 was asked to try to implement an Evolutive Strategy able to solve a black box problem (sort of OneMax) using a diversity promotion method and trying to minimize the fitness calls.

Brief description of the method

Diversity promotion are a set of methods used inside Evolutionary Algorithm designed to maintain diversity within a population, helping to prevent the algorithm from converging prematurely to a single solution and facilitate the exploration of different niches.

Island Model
Is a diversity promotion method that divides the population into subpopulations (islands) and aim to evolve independenty each of them to maintain diversity. Individuals occasionally migrate between island, trying to exchange genetic material among islands. Different islands may converge to different solutions, and migration helps share promising solutions across the islands.

Implementation details

Diversity promotion implemented in this lab works by:

- Splitting the initial population in n subsets
- At each iteration, randomly select one of the subset and evolve it independently
  - Perform a tournament selection among the individual in the subset
  - Given the parent/parents, perform mutation with probability p and recombination with probability $1 - p$
- Select the top individuals inside the generated offpring to build the new population of the subset
- With a probability m perform migration between islands
  - Swap a number of random individuals between all the islands

```python
# evolution loop
for generation in range(ITERATIONS):

  island = islands[randint(0, ISLAND_NUMBER-1)]

  offspring = list()
  for counter in range(OFFSPRING_SIZE):
    if random() < MUTATION_PROBABILITY:
      p = select_parent(island)
      o = mutate(p)
      while random() < MUTATION_REPETITION:
          o = mutate(o)
    else:
      p1 = select_parent(island)
      p2 = select_parent(island)
      o = uniform_xover(p1, p2)
    offspring.append(o)

  for i in offspring:
    i.fitness = fitness(i.genotype)
  island.extend(offspring)
  island.sort(key=lambda i: i.fitness, reverse=True)
  island = island[:ISLAND_SIZE]

  if random() < MIGRATION_PROBABILITY:
    migrate(islands)
```

Since genomes in this case are represented by a sequence of 0 and 1, possible genetic operators are different from the previous work (decimal value genomes -> Gaussian mutation):

- mutation just flip one of the value of the genome (from 0 to 1 and vice versa)
- recombination was performed using two different method
  - one-cut crossover: take two individuals, pick a random index i and take the loci of the first parent from 0 to i, and those from the second parent from i to the end

```
def one_cut_xover(ind1: Individual, ind2: Individual) -> Individual:
  cut_point = randint(0, LOCI_NUMBER-1)
  offspring = Individual(
      genotype=ind1.genotype[:cut_point] + ind2.genotype[cut_point:],
      fitness=None)
  assert len(offspring.genotype) == LOCI_NUMBER
  return offspring
```

- uniform crossover: build the new individual picking each locus from one of the parent uniformly

```
def uniform_xover(ind1: Individual, ind2: Individual) -> Individual:
  offspring = Individual(
      genotype=[choice([ind1.genotype[i], ind2.genotype[i]]) for i in range(len(ind1.genotyp
      fitness=None
  )
  assert len(offspring.genotype) == LOCI_NUMBER
  return offspring
```

Conclusions

I run this implementation with `POPULATION_SIZE=100`, `OFFSPRING_SIZE = 80`, `TOURNAMENT_SIZE = 70`, `ITERATIONS = 5000`, `MUTATION_PROBABILITY = .15`, `MUTATION_REPETITION = .05`, `ISLAND_NUMBER = 7`, `MIGRATION_PROBABILITY = 0.05` and `MIGRATION_SIZE = 5`, but for some reason I wasn't able to solve the problem entirely. On the problem with `size=1`, my program achieved `79.00%` of coverage. I focused more on the implmementation of the diversity promotion than on reducing the fitness call, so I tried to boost it until the delivery date.

Peer review

For this lab, I wrote a review to Salma Aziz-Alaoui - https://github.com/aasalma/Computational-Intelligence.git

LAB10 - Reinforcement Learning

Intro

In lab10 was asked to try to use Reinforcement Learning to devise a tic-tac-toe player.

Brief description of the problem

We all know tic-tac-toe.

Brief description of the method

Reinforcement Learning process involves the Agent observing the current state of the Environment, taking an Action, receiving a Reward and updating its internal policy based on the observed outcomes.

Q-learning
It's a RL technique that maintain internally a table called "Q-table", in which are stored the expected cumulative reward for taking a specific action in a particular state. These values are calculated through the Q-function, that represent the quality of an action given a state: it calculate the expected future reward as the sum of the immediate reward and the maximum Q-value for the next state (**Bellman equation**):

$$Q_{t+1}(s_t, a_t) = Q_t(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right)$$

where:

- α is the learning rate, i.e. lower α lead to slower learning (but maybe gives more precise estimate)
- γ is the discount factor, i.e. how much is taken into account future expected reward

The algorithm, builds this table performing a number of games, trying to balance Exploration and Exploitation:

- Exploration: the Q-learning algorithm, at training time, performs a random action between those available with probability ϵ
- Exploitation: follow and enforce the found policy with probability $1 - \epsilon$

Implementation details

Following https://en.wikipedia.org/wiki/Tic-tac-toe, I implemented an expert agent that I was planning to use as a benchmark (upper bound) of a possible QLearning agent.

Then for the RL agent I implemented a QLearning agent following code provided at https://plainenglish.io/blog/building-a-tic-tac-toe-game-with-reinforcement-learning-in-python.

Choose action

The agent choose an action based on the following code:

```
def choose_action(self, state: State, available_moves: list) -> int:
  # make state hashable to search in the Q-table
  hashable_state = (frozenset(state.x), frozenset(state.o))

  # explore with prob = self.epsilon (train only)
  if np.random.uniform() < self.epsilon:
    return choice(available_moves)
  # exploit
  else:
    # get the Q-value for that state performing different available moves
    Q_values = [self.get_value(hashable_state, action) for action in available_moves]
    # get the max
    max_Q = max(Q_values)
    if Q_values.count(max_Q) > 1:
      best_moves = [i for i in range(len(available_moves)) if Q_values[i] == max_Q]
      i = choice(best_moves)
    else:
      i = Q_values.index(max_Q)
    return available_moves[i]
```

Train loop

This is how Q-learning agent perform the games to update Q-table

```
def Q_train(num_episodes, alpha = .5, epsilon = 0, gamma = .8):
  agent_1 = QLearningAgent(alpha, epsilon, gamma)
  agent_2 = RandomAgent()
  # raw method to adapt exploration-exploitation trade-off
  e_range = np.linspace(1, 0.1, num_episodes)
  game = TicTacToe()
  for step in tqdm(range(num_episodes)):
    # apply desired epsilon at a given index
    agent_1.set_epsilon(e_range[step])

    while not game.game_over:
      state = deepcopy(game.state)
      action = agent_1.choose_action(state, game.available_moves)
      game.make_move(action)

      # if game is not over, make the opponent make a move
      if not game.game_over:
        a2 = agent_2.choose_action(game.state, game.available_moves)
        game.make_move(a2)

      next_state = deepcopy(game.state)
      next_actions = game.available_moves
      reward = game.reward
      agent_1.update(state, action, reward, next_state, next_actions)
    game.reset()
  return agent_1
```

Update
This is how Q-table is updated:

```
def update(self, state, action, reward, next_state, next_available_moves):
  hashable_state = (frozenset(state.x), frozenset(state.o))
  hashable_next_state = (frozenset(next_state.x), frozenset(next_state.o))

  next_Q_values = [self.get_value(hashable_next_state, next_action) for next_action in next_
  max_next_Q = max(next_Q_values) if next_Q_values else 0.0

  Q_value = self.get_value(hashable_state, action)
  self.Q[(hashable_state, action)] = Q_value + self.alpha * (reward + self.gamma * max_next_
```

Conclusions

With `alpha = 0.5`, `gamma = 0.8`, `epsilon_range = [1.0, 0.1]` I achieve good results playing with a random agent -> `wins: 99.42%, ties: 0.58%, loss: 0.00%`, and no loss with the expert agent -> `wins: 0.00%, ties: 100.00%, loss: 0.00%`

Peer review

For this lab, I wrote a peer rewiew to Andrea Galella - https://github.com/andrea-ga/computational-intelligence

QUIXO FINAL PROJECT

Intro

In the final project was required to create players to play to Quixo game.

Brief description of the problem

Quixo is a two-player game composed by a 5x5 board. Each player has their own symbol (as in tic-tac-toe). To win, it's necessary to fill up a row, column, or diagonal with your symbol.
To make a move a player, given their symbol, have to choose a square from the border of the board that has to be neutral or filled up with the player's symbol (opponents symbol square are forbidden to take). Then, choose a legal direction in which to move the square (TOP, BOTTOM, RIGHT, LEFT). The square will take the player's symbol, will be moved to the end of the row/column in the choosen direction and take the final position, sliding all the remaining elements.

Brief description of the method

To solve the task, I've built three different players:

- MinMax player
- QLearning player
- Evolutive Strategy player

MinMax
The MinMax method is a decision-making algorithm designed to determine the best possible move for a player, assuming that the opponent is also playing optimally (minimizing the loss in the worst possible scenario). In this method, the game is represented as a tree, where each node represents a game state, and the edges represent possible moves. Leaf nodes of the tree are evaluated using an evaluation function that assigns a score to each terminal state (win, lose, or draw).
The algorithm works in an adversarial scenario, in which we have one player trying to maximize the reward, and the other one trying to minimize.
The loop used to make a decision is the following:

- begin from the state of the game
- build the tree
  - for each action that can be performed, evaluate possible outcome reaching the bottom
  - backpropagate the information about the outcome: at each level, max player will take the maximum, while min one will take the minimum
- go until the root is reached and choose the action to perform based on the objective of your player

Since my PC wasn't able to travel to the bottom of the tree in admissible time due to lack of computational power, I decided to use pruning mechanism: -

level-based pruning: prune when a maximum level is reached - $\alpha\beta$-pruning: interrupt the exploration of a subtree if an already better solution is present and there's the certainty that all solution in that subtree will be worse

Since pruning level was set to very a low value to obtain an acceptable speed and wasn't enough to reach the end-game node in the tree, evaluation function was improved to give an early estimation of the goodness of an action: the value of a state is evaluated not by who is the winner in that state, but by the maximum number of player's elements in a line minus the maximum number of opponent's elements in a line

Here's the code of the tree generation performed at each decision:

```python
def minmax(self, game: 'MyGame', level: int = 1, alpha = -np.inf, beta = np.inf, player_id:
  available_moves = game.available_moves(player_id)

  # my player plays always as MAX
  if player_id == self.player:
    best = [[], -np.inf]
  else:
    best = [[], +np.inf]

  if len(available_moves) == 0 or game.check_winner() != -1 or level > self.pruning_level:
    #return [[], COMP_RES[self.player][game.check_winner()]]
    return [[], self.get_reward(game)]

  for move in available_moves: # same level nodes
    from_pos = move[0]
    slide = move[1]

    # backup - save row/column
    if slide == Move.LEFT or slide == Move.RIGHT:
      prev_values = deepcopy(game._board[from_pos[1], :])
    else:
      prev_values = deepcopy(game._board[:, from_pos[0]])

    # make a move
    if game._Game__move(from_pos, slide, player_id) == False:
      raise Exception("Invalid move chosen")
    score = self.minmax(game, level+1, alpha, beta, (player_id+1)%2)

    # restore - restore row/column
    if slide == Move.LEFT or slide == Move.RIGHT:
      game._board[from_pos[1]] = prev_values
    else:
      game._board[:, from_pos[0]] = prev_values
```

```
        score[0].append(move)

        # my player plays always as MAX
        if player_id == self.player:
          if score[1] > best[1]:
            best = score  # max value
            alpha = score[1]
          elif score[1] == best[1]:
            best[0].extend(score[0])
        else:
          if score[1] < best[1]:
            best = score  # min value
            beta = score[1]
          elif score[1] == best[1]:
            best[0].extend(score[0])

        if self.soft:
          if alpha >= beta: # <- PRUNE EVEN IF EQUAL
            break
        else:
          if alpha > beta:
            break
      return best
```

QLearning

Dimensionality reduction of state space

Here's a funny story:

I was implementing my QLearning player and I was obtaining stunning results with just `1000` iterations! My agents was winning around `90.20%` of the timeas player 0 and `84.90%` of the time as player 1. Additionally, all of this was happening in just few seconds: unbelivable, right? It was unbelievable actually, because it was all false. I made a little mistake that fools me until few days before the delivery.

In one of my copy-paste session from previous lab, I embraced totally the copy-paste spirit and I forgot to change some details in the function that gave me the hashable state: I was transforming the numpy board in a frozenset! By doing this, all the remaining states were `{-1}`, `{-1,0}` and `{-1,0,1}` and the Q-table was large few KB.

After I fix the bug, the QLearning was performing as expected: really poorly…since quixo has a LOT of possible states (around 3^{12}) it was impossible to catch all of them and derive a good policy!

I decided to report this anyway because it is good to notice that probably, a better encoding of the state that can well summarize the game condition can lead to a good result in considerably less time.

What I thought could be a good state encoding was to represent it as: `[max_p01_row, max_p01_col, p01_1diag, p01_2diag, max_p02_row,` max_p02_col, p02_1diag, p02_2diag] where:

- max_p01_row, max_p01_col, p01_1diag, p01_2diag are respectively index of the row and column with maximum element inside and number of element in the diagonal and counter-diagonal, all of them evaluated for the first player's squares
- max_p02_row, max_p02_col, p02_1diag, p02_2diag the same, but for the second player

Symmetries
I first tried to shrink the state space by implementing a mechanism that takes into account some symmetries

```
def rotate_coord_countclock(self, action, shape):
  _, cols = shape
  from_pos, slide = action
  col, row = from_pos
  rot_row = cols - 1 - col
  rot_col = row
  rot_slide = ROT_SLIDE[0][slide]
  rot_slide += 1
  rot_slide %= 4
  rot_slide = ROT_SLIDE[1][rot_slide]

  return ((rot_col, rot_row), rot_slide)

def rotate_countclock(self, state: np.ndarray, action: tuple[tuple[int,int], Move]) -> tuple
  return (np.rot90(state), self.rotate_coord_countclock(action, state.shape))

def get_value_wsymmetry(self, state, action):
  hashable_state = QPlayer.state_to_set(state)
  if (hashable_state, action) in self.Q:
    return hashable_state, action, self.Q[(hashable_state, action)]
  #  90 rotation
  r_state, r_action = self.rotate_countclock(state, action)
  r_hashable_state = QPlayer.state_to_set(r_state)
  if (r_hashable_state, r_action) in self.Q:
    return r_hashable_state, r_action, self.Q[(r_hashable_state, r_action)]
  # 180 rotation
  r_state, r_action = self.rotate_countclock(r_state, r_action)
  r_hashable_state = QPlayer.state_to_set(r_state)
  if (r_hashable_state, r_action) in self.Q:
    return r_hashable_state, r_action, self.Q[(r_hashable_state, r_action)]
  # 270 rotation
  r_state, r_action = self.rotate_countclock(r_state, r_action)
  r_hashable_state = QPlayer.state_to_set(r_state)
  if (r_hashable_state, r_action) in self.Q:
    return r_hashable_state, r_action, self.Q[(r_hashable_state, r_action)]
  # not present
  self.Q[(hashable_state, action)] = 0.0
  return hashable_state, action, self.Q[(hashable_state, action)]
```

then I gave up and focused on the manually reduced version of the algorithm.

Evolutionary Algorithm
Basically, I used a set of 44 parameters, each representing a possible move, to be tweaked (i.e. Gaussian mutated) at each iteration. Each parameters describe the probability of doing that move. The algorithm works as previous lab on evolutive strategy, with some adaptation to the problem.

Collaboration

For the final project, I shared some idea with my collegue Alberto Foresti - s309212 - https://github.com/AlbertoForesti/computational_intelligence

Conclusions

MinMax
It's the slowest in decision making, but it achieve best results: with `pruning_level=2`, as `player=0` => `win_rate=100.00%`, as `player=1` => `win_rate=96.00%`.

QLearning
QBug
With the initial wrong state encoding, training was really fast and `win_rate=90.20%` as player 0 and `win_rate=84.90%` as player 1, with only `iterations=1000`, but it highly depends on initialization: some lucky run reached that level of `win_rate`, some other performed very poorly. Parameters used in this first try were: `alpha=0.3`, `epsilon_range=[1.0,1.0]`, `gamma=0.99`. I trained different models and picked the best one as final player.
QReduced
For the reduced-space variant, I tried different implementations:

- `red1` -> `state=[max_p01_row, max_p01_col, p01_1diag, p01_2diag, max_p02_row, max_p02_col, p02_1diag, p02_2diag]`
  too large, must shrink more
- `red2` -> `state=[max_p01_row, max_p01_col, max_p01_diag, max_p02_row, max_p02_col, max_p02_diag]`
  a little more...
- `red3` -> `state=[max_p01_row, max_p01_col, p01_1diag, p01_2diag]`
  Here we are! I first trained with max exploration rate, i.e. `init_epsilon=1.0`, `final_epsilon=1.0`, for `100K` iterations and then with `init_epsilon=0.8`, `final_epsilon=0.4` for `30K` iterations with `alpha=0.3` and `gamma=0.99` obtaining `win_rate=71.00%` as player 0 and `win_rate=65.00%` as player 1.

QSymmetry
I abandoned the idea to consider symmetry: I implemented a symmetry detection system (not completely, I've considered just some of the symmetries), but it seemed that even `100K` iteration wasn't enough. I focused on searching a better state representation, giving more attention on efficiency in training phase rather than brute forcing all possible states (avoid doing infinite `1M` iterations run)

Evolutionary Algorithm
I believe that I've implemented my algorithm with some performance issues, so I wasn't able to perform any long training run (`10K` runs gave me an expected time of 350 hours...). Training for `100` (even few are enough since the fitness value got stuck at `0.666` at `38-th`) iteration was enough to make my player achieve a `win_rate=62.60%` as player 0, and `win_rate=57.18%` as player 1.
Probably some other mechanism, would enhance the performance of my EA.
I also explored the parameter list to check what move make the agent win on average, here are the `top-5`:

```
# moves are stored as (column, row)
0.07772431367010198
((3, 4), <Move.RIGHT: 3>)
0.07651940154722346
((2, 0), <Move.BOTTOM: 1>)
0.07503089727817207
((1, 4), <Move.LEFT: 2>)
0.06988633953007747
((4, 2), <Move.TOP: 0>)
0.056676266207186514
((4, 2), <Move.BOTTOM: 1>)
```