Computational Intelligence - Activity Report

Borella Simone
s317774
borellasimone00@gmail.com

January 30, 2024

Abstract

The following text describes activities related to the "Computational Intelligence" course held by professor Giovanni Squillero at Politecnico di Torino during the academic year 2023/2024. Implementations and material can be found in the GitHub repository https://github.com/SimoneBorella/computational-intelligence.git.

1 Lab1 - Set covering with A*

1.1 Introduction

Given a finite set U of n elements and a collection S of m subsets of U, the goal behind the set covering problem is to find the smallest subset of S such that the union of its elements covers all elements in U.

$$\begin{aligned} &\text{Minimize} \quad |C| \\ &\text{Subject to} \quad \bigcup_{S_i \in C} S_i = U, \quad \text{where} \quad S_i \in S \end{aligned} \tag{1}$$

The A* is a searching algorithm aimed at finding the shortest path between an initial and a final state. The algorithm estimates a total cost from the initial to the final state for each state reached.

1.2 Implementation

For each state reached, a cost function and an heuristic function are computed.

The cost function represents the cost to reach the current state from the initial state, which in this case is the actual cardinality of the covering set as shown in Equation 2.

$$\text{cost}(C) = |C| \tag{2}$$

The heuristic function estimates the cost to reach the final state, starting from the actual state. Two heuristics were proposed:

- **Heuristic1:** This heuristic compute for each set the number of uncovered elements and then returns the number of state which sum of uncovered elements is greater or equal to the number of uncovered elements of the current covering set.

$$h1(C) = min(|S'|) \quad \text{where} \quad S' \subseteq S, \quad \sum_{S'_i \in S'} |S'_i| \geq |U \setminus \left(\bigcup_{S_i \in C} S_i \right)| \tag{3}$$

- **Heuristic2:** This heuristic is following a similar approach of the first heuristic but returns the minimum number of sets needed to fill the actual covering set ensuring that the sets will complete it, including all elements. So for each set considered the number of uncovered elements is recomputed and the next set to consider is based on the last choice.

$$h1(C) = min(|S'|) \quad \text{where} \quad S' \subseteq S, \quad \left(\bigcup_{S_i \in C} S_i \right) \bigcup \left(\bigcup_{S'_i \in S'} S'_i \right) = U \tag{4}$$

1.3 Results

These heuristics where tested over 500 experiments with $|S| = 100$ number of sets randomly generated and $|U| = 20$ total elements.

Here are represented the average number of steps needed to solve the problem with success:

- Heuristic1: 24.542 steps
- Heuristic2: 13.042 steps

2 Lab2 - Nim game with genetic algorithm

2.1 Introduction

The Nim game is a two player game consisting in removing objects form a certain number of rows containing a certain number of objects. The player who takes the last object lose. The player in turn can chose only one row and must remove at least one element.

A genetic algorithm (part of the class of evolutionary algorithms) is a type of optimization algorithm inspired by the principles of natural selection and genetics. It is used to find approximate solutions to optimization and search problems, especially in complex and high-dimensional spaces.

2.2 Implementation

Genome

The genome structure in this algorithm is a simple set of rules which the agent will apply with a certain probability. So the genome is represented as an array of weights between 0 and 1.

The rules applied are:

- Heap selection:
    1. Random choice
    2. Fattest heap
- Object selection:
    1. Random selection
    2. Odd selection
    3. Prime number selection
    4. Leave one strategy

Here is a representation of weights in an array:
[<rand_heap>, <fat_heap>, <rand_obj>, <odd_obj>, <prime_obj>, <leave_obj>]

Fitness

The fitness function is evaluated as the mean of the number of wins making the agent play, for a certain number of matches, against a random agent among pure_random strategy and optimal strategy. The two strategies are selected with a probability given by the difficulty set. Proceeding in generations the difficulty becomes higher and the probability to select the optimal strategy increase, allowing individuals of the population to improve against the best strategy.

Mutation

The mutation is performed with a mutation probability on each individual of the population selecting randomly one element of the genome and then adding a Gaussian random value (Gaussian mutation).

Crossover

The crossover between two genomes is performed creating a new genome iterating each element of the two parents and chosing randomly one of the elements of the parents (Uniform crossover).

Parent selection

Three parent selection strategies are tested:

- Tournament selection is adopted taking from a sample of the population the fittest individuals
- Roulette wheel selection is adopted taking parents in a pseudo random way
- 'Best selection' taking the n best parents from the population.

Results

Simulation data:

- N_GEN = 200
- POPULATION_SIZE = 50
- N_PARENTS = POPULATION_SIZE//3
- MUTATION_RATE = 0.15
- FITNESS_MATCHES = 200
- TOURNAMENT_SIZE = 25

Figure 1 describe the evolution of the population. Note that the difficulty is increasing, and so going over generations, during fitness evaluation, the probability to have a match against the optimal strategy is increasing and at the end the fitness is evaluated solely with the optimal strategy setting an upper bound to 0.5 winning rate, which is an optimal strategy agent against an other optimal strategy agent.

Figure 1: Evolution of the population

The best genome obtained is $[0, 1, 0, 0, 0, 1]$, which means that the best strategy found is to leave one object from the fattest heap.

3 Lab3 - Local optima search with genetic algorithm

3.1 Introduction

The objective is to find local optima using the minimum number of fitness calls.

3.2 Implementation

Genome

The genome is a bitstring of 1000 loci.

Fitness

The fitness function is provided as a black box in function of a problem instance (1, 2, 5 or 10 in this case).

Diversity

Diversity between two individuals is computed as hamming distance normalized in the range [0, 1].

Mutation

The mutation is performed with a mutation probability on each individual of the population selecting randomly one element of the genome and then switching it (Bit flip mutation).

Crossover

The crossover between two genomes is performed creating a new genome iterating each element of the two parents and chosing randomly one of the elements of the parents (Uniform crossover).

Parent selection

A tournament selection is adopted taking from a sample of the population the individuals with the best score in function of the fitness and diversity.

$$score = k * fitness + (1 - k) * diversity \tag{5}$$

Termination

Termination is reached with a convergence criterion (fitness change ¡ 0.01 in 100 generations).

3.3 Results

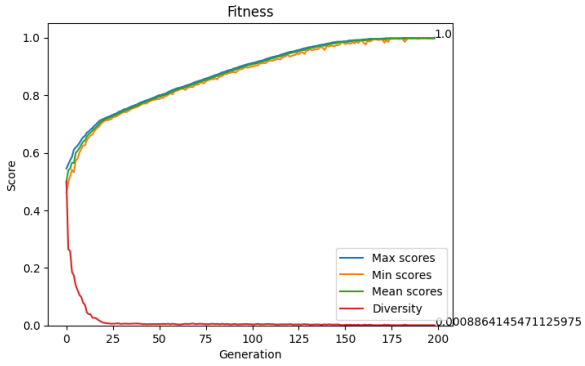
Simulation data:

- N_PARENTS = POPULATION_SIZE//3
- MUTATION_RATE = 0.5
- TOURNAMENT_SIZE = POPULATION_SIZE//3
- K = 0.7

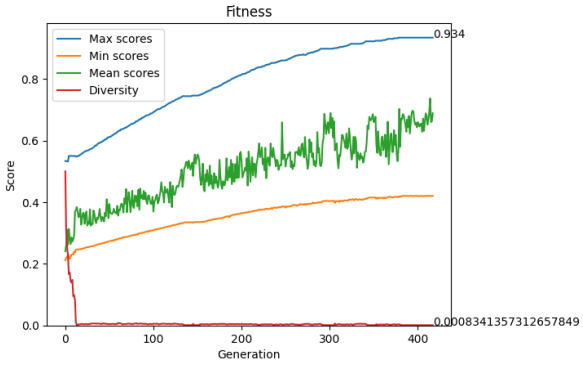
Problem instance	Generations	Fitness calls	Fitness reached
1	199	35820	1.0
2	418	68340	0.934
5	107	19220	0.431
10	108	17600	0.337

Table 1: Result table

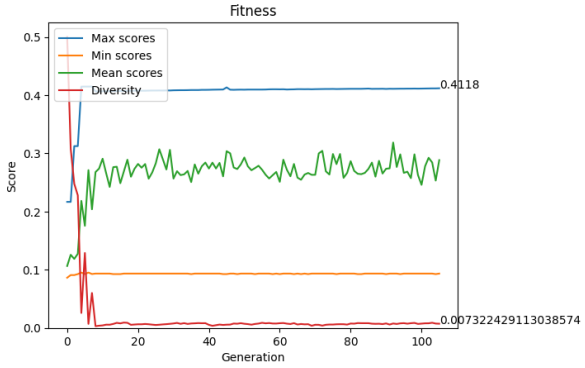
The figures below describe the growing of the population for each problem size (1, 2, 5 and 10).



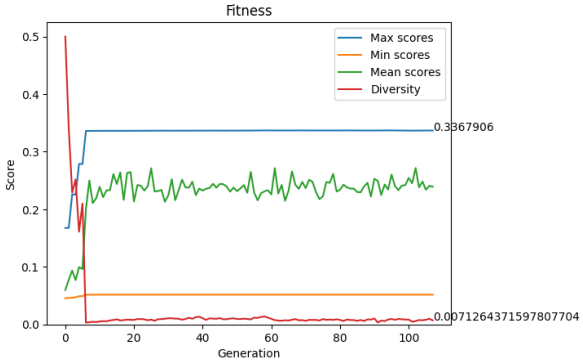
(a) Problem size 1



(b) Problem size 2



(c) Problem size 5



(d) Problem size 10

Figure 2: Evolution of the population

4 Lab 4 - Tic Tac Toe with Reinforcement learning

4.1 Introduction

The objective is to find an optimal agent for playing tic tac toe with reinforcement learning techniques.

4.2 Implementation

Q-learning

Q-learning is a model-free reinforcement learning algorithm. This algorithm deals with learning a policy, which is a strategy for making decisions, in a Markov decision process (MDP). In a MDP the future state depends only on the current state and action, not on the sequence of events that preceded them. A reward is given every time, given a state, an action is selected to go to the next state. The goal of Q-learning is to learn an optimal policy maximising the cumulative reward.

Environment

The gymnasium library is exploited to create a custom rl environment.

State The state is represented as a numpy array with shape 3x3 filled with 0 (empty), 1 (agent player), 2 (opponent player).

Action An action is represented by an integer number indicating the cell in which the agent wants to put its X.

Rewards Given a state and an action the following rewards are assigned:

- Win reward: 10
- Lose reward: -10
- Draw reward: 3

Opponent strategies and first turn The environment class takes an argument to pass a pool of opponent startegies which are chosen randomly and an optional argument to decide the whose is the first turn (random if not set).

Opponent Strategies

Random strategy Selects random moves.

Magic strategy (optimal) It exploit a magic matrix with the property of having the sum of each row column and diagonal equal to 15.

It act randomly until the state contains two cells of the optimal agent.

Then tries to find an action that allow the agent to win, evaluating for each couple of cells of the agent the sum of the corresponding values in the magic matrix subtracted from 15 and then checking, if the values is in the range [1, 9], if the corresponding cell in magic matrix is free and filling that cell.

If no winning moves are possible then tries to find an action that block the opponent player, performing the same operation but considering opponent cells.

Agent strategy The agent strategy is used during training to make the agent learn playing against itself.

Q-learning agent

The agent class holds q_table values which contain the action-value function values for each couple of action-state. Since not all states are feasible for tic tac toe the q_table is a defaultdict with state as key and an array of 0 values for each action.

Q table update In the training phase the q_table is updated with the following rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big[(R_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a')) - Q(s_t, a_t)\Big] \tag{6}$$

s

where $Q(s_t, a_t)$ is the value of taking action a in state s_t, R_{t+1} is the immediate reward, α is the learning rate, γ is the discount factor, and s_{t+1} is the next state.

Hyperparameters $\alpha = 0.1$, $\gamma = 0.9$.

Action selection

Baed on the exploration rate ϵ two action selection methods are used:

- Random selection
- Best action selection (action which maximises the future reward)

Training

The following training steps have been done:

- Training with opponent strategies = (random_strategy)
- Retraining with opponent strategies = (random_strategy, agent_strategy, strategy)
- Retraining with opponent strategies = (agent_strategy, magic_strategy)

Each training step is performed evaluating 500000 episodes.

The ϵ parameter is evaluated for each episode in compliance with GLIE (Greedy in the Limit with Infinite Exploration) theorem.

$$\epsilon_k = \frac{b}{b+k} \tag{7}$$

where ep is the current episode number and b is tuned to have $\epsilon = 0.1$ at 90% of the training and leaving that value for the remaining 10%.

Results

Tests are evaluated on 10000 episodes.

Opponent strategy	Initial turn	Win rate	Lose rate	Draw rate
random_strategy	random	98.12%	0.00%	1.88%
random_strategy (first move)	QLearningAgent	98.13%	0.00%	1.87%
random_strategy	OpponentAgent	91.49%	0.00%	8.51%
magic_strategy	random	74.89%	0.00%	25.11%
magic_strategy (first move)	QLearningAgent	75.15%	0.00%	24.85%
magic_strategy (opponent first move)	OpponentAgent	6.02%	0.00%	93.98%

Table 2: Results table

5 Exam project - Quixo optimal strategy

5.1 Introduction

Quixo is known for its complexity and the multitude of potential moves at each turn, making it a subject for the application of artificial intelligence techniques. My objective is to analyze and present different approaches and methodologies that can be employed to develop an agent capable of making optimal decisions in the context of Quixo.

To achieve this objective, we experimented with three distinct approaches:

- MiniMax with alpha-beta pruning
- Deep Q-learning
- Value iteration with backward induction

Each approach brings its own set of strengths and challenges to the table, offering a diverse perspective on the problem.

Since, as explained in results, Quixo 5x5 is a draw game, which means that if both player agents are optimal neither player can win, in testing phase the initial turn is not specified, while is specified in Quixo 4x4 testing, which is a first-player-win game.

5.2 MiniMax with alpha-beta pruning

5.2.1 Introduction

The MiniMax algorithm is a decision-making algorithm used in two-player games with perfect information. Its primary objective is to determine the optimal move for a player, assuming that the opponent will also make optimal moves. The algorithm is based on the principle of minimizing the possible loss for a worst-case scenario.

5.2.2 Implementation

The game states are represented as nodes of a tree and each level of the tree corresponds to a turn in the game. For each leaf node in the tree a value is assigned based on an heuristic evaluation function that estimates the desirability of that game state for the current player.

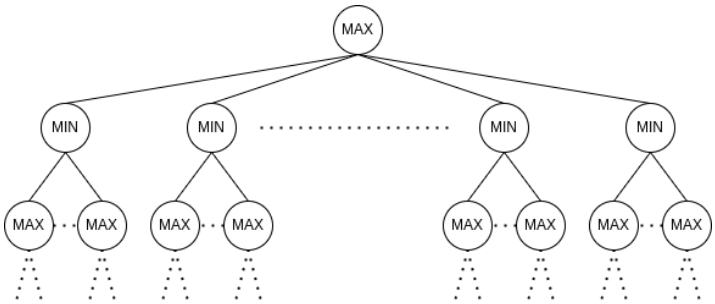


Figure 3: MiniMax state tree

The heuristic evaluation function is described as:

$$H(N, C_{max}, O_{max}) = N^{C_{max}} - N^{O_{max}} \tag{8}$$

where N is the board size, C_{max} is the maximum number of current player symbols in the same line and O_{max} is the maximum number of opponent player symbols in the same line, where a line can be an horizontal, vertical or diagonal line.

Starting from the root node, the algorithm recursively explores the tree by considering all possible moves at each level. At a Max node, the algorithm selects the move that leads to the maximum value among its child nodes. At a Min node, the algorithm selects the move that leads to the minimum value among its child nodes.

Alpha-beta pruning is an optimization technique used in the minimax algorithm to reduce the number of nodes evaluated in the search tree. During the tree recursive visit two parameters are maintained, alpha and beta which respectively represent the maximum value that the maximizing player has found so far at any choice and the minimum value that the minimizing player has found so far at any choice. During the traversal of the tree, when the maximizing player encounters a node with a value greater than or equal to beta, it realizes that the minimizing player)will never choose this path because the opponent has a better option and the entire subtree can be pruned. Similarly this happens when a minimizing player encounters a node with a value less than or equal to alpha.

5.2.3 Results

Since the action space is wide, having at most $4 \cdot N$ possible moves for each state, where N is the board size, the tree will have about $(4N)^{tree\_height}$ leaf nodes to compute.

Tree height	Leaf nodes
2	256
3	4096
4	65536

Figure 4: Tree leaf nodes with $N = 4$

Tree height	Leaf nodes
2	400
3	8000
4	160000

Figure 5: Tree leaf nodes with $N = 5$

To be computational efficient, the maximum depth I took in account is 3, since with a higher depth the agent is extremely slow.

Here are shown some testing results in a 5x5 table ($N = 5$):

Agent	Opponent Agent	Win rate	Lose rate	Draw rate
MiniMaxAgent(2)	RandomAgent	100.0%	0.0%	0.0%
MiniMaxAgent(2)	MiniMaxAgent(2)	49.0%	51.0%	0.0%
MiniMaxAgent(3)	MiniMaxAgent(2)	100.0%	0.0%	0.0%

Table 3: Agent performance comparison with $N = 5$

Here are shown some testing results in a 4x4 table ($N = 4$):

Agent	Opponent Agent	Initial player	Win rate	Lose rate	Draw rate
MiniMaxAgent(2)	RandomAgent	0	100.0%	0.0%	0.0%
MiniMaxAgent(2)	RandomAgent	1	99.0%	1.0%	0.0%
MiniMaxAgent(2)	MiniMaxAgent(2)	0	0.0%	0.0%	100.0%
MiniMaxAgent(2)	MiniMaxAgent(2)	1	0.0%	0.0%	100.0%
MiniMaxAgent(3)	MiniMaxAgent(2)	0	100.0%	0.0%	0.0%
MiniMaxAgent(3)	MiniMaxAgent(2)	1	0.0%	0.0%	100.0%

Table 4: Agent performance comparison with $N = 4$

5.3 Deep Q-learning

5.3.1 Introduction

Quixo game can be easily seen as a Markov Decision Process (MDP), which means that given a state the action that maximizes the future reward depends only on the current state.

Deep Q-Learning is a reinforcement learning technique that combines Q-Learning, an algorithm for learning optimal actions in an environment, with deep neural networks. It aims to enable agents to learn optimal actions in complex, high-dimensional environments. By using a neural network to approximate the Q-function, Deep Q-Learning can handle environments with large state spaces.

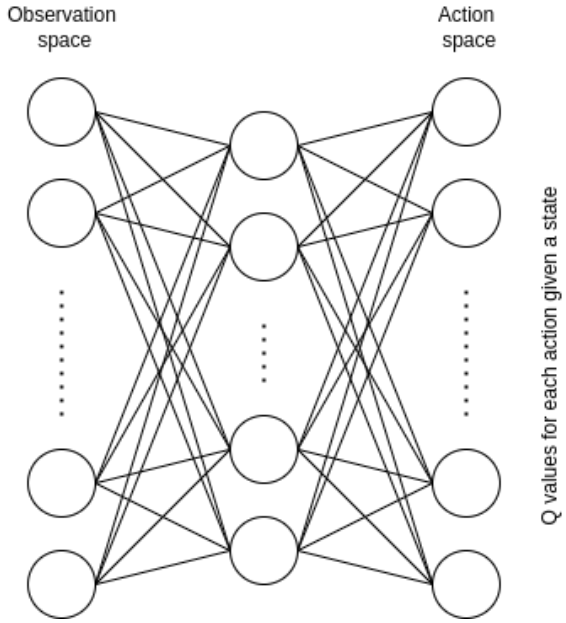


Figure 6: Q-network

Q-learning aims to learn the action-value function, denoted as Q-function, which represents the expected cumulative reward of taking an action in a particular state and following a certain policy. Q-learning is based on the Bellman equation, which expresses the relationship between the value of a state-action pair and the values of its successor states.

The update rule is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \Big[(R_{t+1} + \gamma \cdot \max_{a'} Q(s_{t+1}, a')) - Q(s_t, a_t)\Big] \tag{9}$$

5.3.2 Implementation

Since the observation space counts $3^{N \cdot N}$ possible state combinations and the action space counts a maximum of $4 \cdot N$ possible actions, a tabular Q-learning would be too expansive in terms of memory occupation, since for each state there should be a number of action-values equal to the number of actions.

Assuming that each value is a 4 byte integer the occupied memory is equal to:

$$Memory\_occupation = observation\_space \cdot action\_space \cdot value\_occupation = 3^{N \cdot N} \cdot 4 \cdot N \cdot 4 \quad bytes$$

which is equal to 656.8 MB with $N = 4$ and 15.4 TB with $N = 5$.

This is why a deep Q-learning solution with a neural network for function approximation is necessary.

The neural network is composed by three fully connected layers, the input layer (state input) with $N \cdot N$ neurons, an hidden layer with 256 neurons and an output layer (action-value output) with $4 \cdot N$ neurons.

The reward function is pretty simple, if at the end of an episode the agent wins the reward is positive, otherwise 0:

$$R(s, a) = \begin{cases} 200 - episode\_number & \text{agent wins} \\ -(200 - episode\_number) & \text{agent lose} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

In this way theoretically the agent strategy will converge to a policy derived by the action value function which objective is to win in the minimum number of steps.

Choosing the appropriate value for the exploration-exploitation tradeoff parameter ϵ is a critical decision in reinforcement learning. This parameter is commonly used in strategies such as epsilon-greedy policies, which determine how the agent balances between exploring new actions and exploiting the current best-known actions. In this case a GLIE (Greedy in the limit of infinite exploration) ϵ parameter is chosen with the following equation:

$$\epsilon_k = \frac{b}{b + k}, \quad b = k \cdot \frac{\epsilon_{final}}{1 - \epsilon_{final}} \tag{11}$$

where k is the current episode number.

The idea behind GLIE is to ensure that the agent explores sufficiently during the initial stages of learning when knowledge about the environment is limited, gradually becoming more greedy as it accumulates more experience and converges toward an optimal policy.

5.3.3 Results

Several experiments were done changing reward function and hyperparameters.

Below are explained the training steps followed using both the reward function described in Equation 10:

1. 100000 episodes training against RandomAgent
2. 100000 episodes training against RandomAgent and MiniMaxAgent(2)
3. 100000 episodes training against MiniMaxAgent(2)

Testing results:

Agent	Opponent Agent	Win rate	Lose rate	Draw rate
DeepQLearningAgent	RandomAgent	87.0%	13.0%	0.0%
DeepQLearningAgent	MiniMaxAgent(2)	0.0%	100.0%	0.0%

Table 5: Agent performance comparison with $N = 5$

Agent	Opponent Agent	Initial player	Win rate	Lose rate	Draw rate
DeepQLearningAgent	RandomAgent	0	84.0%	16.0%	0.0%
DeepQLearningAgent	RandomAgent	1	67.0%	33.0%	0.0%
DeepQLearningAgent	MiniMaxAgent(2)	0	0.0%	100.0%	0.0%
DeepQLearningAgent	MiniMaxAgent(2)	1	0.0%	100.0%	0.0%

Table 6: Agent performance comparison with $N = 4$

5.4 Value iteration with backward induction

5.4.1 Introduction

The game Quixo, played on a 5x5 grid, involves two players which aim is to align five identical symbols. This game requires innovative strategies to win, and here are described the strategies exploration steps to find an optimal agent strategy capable of solving Quixo game in an efficiently way.

The solution proposed here is based on the findings presented in the cited paper [ST20]. The paper propose a solution for finding an optimal strategy, assuming perfect players, treating the Quixo game as a Markov Decision Process (MDP) and using the well known Value Iteration algorithm assigning an outcome (Win, Loss, Draw) for each state of the game.

5.4.2 Implementation

Here the objective is to find the optimal state-value function (state-values are also called outcomes here) of the MDP.

Implementation details are referred to Quixo 5x5 ($N = 5$).

A state of the game consists of a board and an active player (X player is assumed to be the active player). The initial state is the empty board. A state is terminal if its board contains a line of Xs or Os tiles. The children of a given state are all states obtained by a move of the active player (and then swapping the Xs tiles with Os tiles to keep the X active player representation). A terminal state has no children since the game is over and there is no valid moves. The parents of a state are defined analogously. The set of states and parent-child relations induce the game graph of Quixo, which is neither a tree nor an acyclic graph.

Each state has a state value, also called outcome which can be either active-player-Win, active-player-Loss, or Draw. For brevity, the active-player part is omitted, and a Win (resp. Loss and Draw) state denotes a state whose outcome is Win (resp. Loss and Draw).

For terminal states the outcome is defined as follows (take in account that every time a move is taken the active player is swapped):

- Win if there is a line of Xs
- Loss if there aren't Xs lines and there is a line of Os

For non-terminal states, the outcome is inductively defined as follows:

- Win if there is at least one Loss child
- Loss if all children are Win
- Draw otherwise

The most natural representation of a state in memory is to use an array of 25 elements where each entry takes values from the set 0, 1, 2 to map empty, X, O. Assuming a typical 1-byte char element, such a representation requires 25 bytes per state.

To reach a more compact state representation, the sequence of 25 numbers could be seen as a single number written in ternary basis. Hence each state corresponds to a unique integer in $\{0, \ldots, 3^{25} - 1\}$. Since $\log_2 3^{25} \approx 40$ bit, a single 8 byte variable is sufficient to store a state.

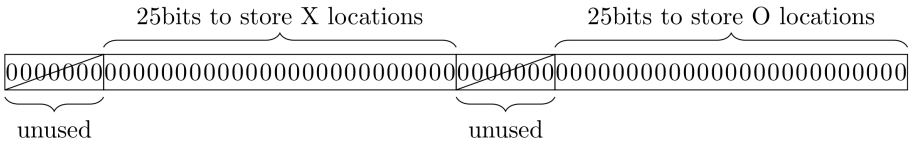


Figure 7: State representation

This representation offers some decisive advantages. It enables very fast computation of all basic operations with bitwise operators.

In order to strongly-solve the game, we need to record the outcome of all possible states. Three possible outcomes (Win, Loss or Draw) means that 2 bits are necessary to store each outcome. Using a typical state-value associative array requires at least $64bits + 2bits$ per entry, which sums up to more than $6.5TB$. It is possible to enumerate all possible states in a pre-determined order creating a bijection between the set of all 3^{25} states and the set of natural numbers $\{0, \ldots, 3^{25} - 1\}$ storing the outcomes in a giant bit array. Again, $2bits$ per entry yields a total size of $2 \cdot 3^{25} = 197GB$.

The solution to the memory occupation is storing only part of the states outcomes in RAM using backward induction. Introducing the concept of class $C_{x,o}$ as the class of states having x Xs and o Os, to evaluate a generic state $C_{x,o}$ we just need to know the outcomes of $C_{o,x}$ and $C_{o,x+1}$, since an available move could move an empty tile that becomes X or could move an X tile, swapping the players we obtain the two possible state classes. The largest class is $C_{8,8}$ which contains $\binom{25}{8} \cdot \binom{17}{8} \approx 2.6 \cdot 10^{10}$ states, which corresponds to $6.1GB$, which multiplied by 4 (class outcomes, reverse class outcomes, next class outcomes, reverse next class outcomes) reaches an approximatively maximum memory occupation of $24.4GB$.

Another hidden problem here is how to create a bijection between states of class $C_{x,o}$ and the set of natural nummbers $\{0, \ldots, \binom{25}{x} \cdot \binom{25-x}{o} - 1\}$.

In order to do that the 64-bit state variable are split into two 32-bit variables. For the variable representing the location of the Os, we remove the digits where Xs are located, and then shift remaining digits to the right as shown in Figure 8.

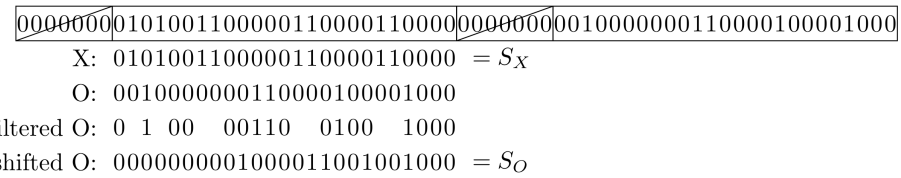


Figure 8: Sx, So extraction from state representation

The index of the state whitin its class is defined as:

$$class\_state\_index = ord(S_x) \cdot \binom{25-x}{o} + ord(S_o) \tag{12}$$

where $ord(\cdot)$ is an order function that returns the order of a number among numbers with the same population count (Hamming weight).

In order to obtain fast computation $pop(\cdot)$ and $ord(\cdot)$ functions are precomputed.

I'm reporting here the algorithms used in this solution.

Algorithm 1 shows the basic Value Iteration algorithm which is not feasible to run because of the amount of memory needed to store all states outcome.

Algorithm 1 Value Iteration

```
for all states s do
    if there is a line of Xs in s then
        outcome[s] ← Win
    else if there is a line of Os in s then
        outcome[s] ← Loss
    else
        outcome[s] ← Draw
    end if
end for
repeat
    for all states s such that outcome[s] = Draw do
        if at least one child of s is Loss then
            outcome[s] ← Win
        else if all children of s are Win then
            outcome[s] ← Loss
        end if
    end for
until no update in the last iteration
```

Algorithm 2 shows Value Iteration algorithm using backward induction making feasible to compute outcomes of states belonging to class $C_{x,o}$ with the knowledge of outcomes of states belonging to classes $C_{o,x}$ and $C_{o,x+1}$.

Algorithm 2 Backward induction with Value Iteration

```
for n = 25 to 0 do
    for x = 0 to n/2 do
        o ← n − x
        if n < 25 then
            Load outcomes of classes C_{x,o+1} and C_{o,x+1}
            Compute outcomes of classes C_{x,o} and C_{o,x} using VI
            Save outcomes of classes C_{x,o} and C_{o,x}
            Unload all outcomes
        end if
    end for
end for
```

A parallelization technique is involved to speed up the computation. In particular for each time there is a computation over all states of a class, the states in that class are divided by the number of threads and each is responsible of its states. A particular attention is given to mutual access to shared resources using mutexes and conditional variables.

5.4.3 Results

The program was written in C++ and was run on a Ubuntu 22.04LTS machine equipped with 32GB of RAM and powered by a 16-core Intel Core i9 CPU. The computation for Quixo 4x4 lasted about 30 minutes, whereas for Quixo 5x5, it took about 60 hours (2 days and a half). Quixo 4x4 has produced about 10.3 MB of data while Quixo 5x5 produced 197 GB of data.

Here are shown some testing results in a 5x5 table ($N = 5$):

Agent	Opponent Agent	Win rate	Lose rate	Draw rate
QuixoIsSolvedAgent	MiniMaxAgent(2)	18.2%	0.0%	81.8%
QuixoIsSolvedAgent	MiniMaxAgent(3)	11.0%	0.0%	89.0%
QuixoIsSolvedAgent	QuixoIsSolvedAgent	0.0%	0.0%	100.0%

Table 7: Agent performance comparison with $N = 5$

Here are shown some testing results in a 4x4 table ($N = 4$):

Agent	Opponent Agent	Initial player	Win rate	Lose rate	Draw rate
QuixoIsSolvedAgent	MiniMaxAgent(2)	0	100.0%	0.0%	0.0%
QuixoIsSolvedAgent	MiniMaxAgent(2)	1	27.6%	72.4%	0.0%
QuixoIsSolvedAgent	MiniMaxAgent(3)	0	100.0%	0.0%	0.0%
QuixoIsSolvedAgent	MiniMaxAgent(3)	1	23.2%	76.2%	0.6%
QuixoIsSolvedAgent	QuixoIsSolvedAgent	0	100.0%	0.0%	0.0%
QuixoIsSolvedAgent	QuixoIsSolvedAgent	1	0.0%	100.0%	0.0%

Table 8: Agent performance comparison with $N = 4$

These results confirm all the results explained in the cited paper [ST20], and so Quixo 4x4 is a Active-Player-Win game while Quixo 5x5 is a Draw game, which means that two optimal players will play endlessly.

6 Reviews

6.1 Lab 2

Giuseppe Nicola Natalizio - s305912

Hello there, Simone! I liked the way you report is written, it's very detailed and easy to follow and the addition of graphs made your observations easy to understand. The code is cleanly written and easy to understand, there's only one thing I noticed I'd like to point out. The generate_random_agent function returns an individual with all the weights set to 0.5 rather than random, this means that when you initialise the population all the individuals have the same genome, this is partially compensated by the mutation, however, with such a low mutation chance we can expect most of the population to have the same genome. I don't think this causes an issue in this particular problem, but in general it could lead to an inadequate exploration of the solution space. I hope this review was useful to you and good luck on your next labs.

Caretto Michelangelo - s310178

Hello Simone, talking about rules, I liked the way you splitted the two different choices : 1-Which is the best row for taking object?, 2-How many objects shall i take? Thanks to this choice, we can explore more the Nim problem, and answer to a question, can the choice of row and the number of object can be uncorrelate? And the answer by looking at graph seems "Yes" ,because you manage to reach almost 50% of winrate against the optimal solution, ... but not everything is as it seems , cz with a more curiosity you would have notice that "optimal" function in reality is not that optimal. So I very liked this different approch for the Nim games, and i liked too the way you implemented the GA algortihm, because all parameter are balanced up to me. I hope you'll enjoy this review and have fun playing CI. Bye bye!

6.2 Lab 3

Martini Martina - s306163

I found the work about local search very interesting, particularly the choice of the hamming distance for the diversity evaluation. I appreciated the plots for all the problem instances. Still, I would like

to suggest you a better representation of the graphs: in problem instance 1 the lines are very close to one another so they can result ambiguous. Moreover, the absence of the grid does not let the reader fully evaluate the results. The last advice I would like to leave to you regards the choice of the crossover mechanism: I would introduce a crossover rate, in order to introduce more randomness to your algorithm. In conclusion, I found your work extremely complete. Well done!

Samaneh Gharehdagh Sani - s309100

Dear Simone, I've reviewed your genetic algorithm code and would like to commend your well-structured implementation, particularly in the areas of fitness evaluation, mutation, crossover, and parent selection. However, I have a few suggestions for improvement: Local Search Techniques: The manuscript title suggests the use of local search techniques, but they aren't clearly integrated. Clarification on this aspect would align the content with the title's expectations. Code Clarity: Adding more in-line comments explaining key code sections could improve readability and understanding for others. Comparative Analysis: A comparison with other algorithms or benchmarks could provide a clearer context for evaluating your GA's efficiency and effectiveness. Overall, your work shows great promise. Enhancing these aspects could significantly improve the manuscript's depth and clarity. I look forward to seeing your continued progress in this area. Best regards, Samaneh Gharehdagh Sani

Florentin-Cristian Udrea - s319029

Hi Simone, I found you code very clear! One particular idea I liked is the early stopping the GA algorithm by looking at generation-by-generation improvement on the population. I saw you decided that, if there is no improvement greater than a certain threshold for a fixed number of generation, you stop the algorithm. I guess in this solution may have worked, but I think generally finding this threshold can be hard. Sometimes what happens is that after a long plateau (a vast region in the fitness function where all the solution have more or less the same fitness) there is the global optimum. Also if the fitness ranges are very small (for example between 0 and 0.1) in a certain problem you would always stop early. I think a better idea would be stopping if there is no improvement for a fixed number of generations. This would solve the problem of getting stuck in a local optima and uselessly iterating, while also getting over plateaus without problem. I hope I was useful !

6.3 Lab 4

Caretto Michelangelo - s310178

Hi Simone, i really liked your Q-Learning Implementation. I liked the way that you implemented for choosing an action at training time, i think your strategy it's really good because it seems like a baby that at the start is doing random thing and then something rational. The results table is not clear and readable next time try to specify for eg. the difference between magic matrix -mm(first move)-mm(opponent firstmove). Anyway good job!

References

[ST20] Sébastien Tixeuil Yasumasa Tamura Satoshi Tanaka, Francois Bonnet. Quixo is solved. 2020.