Final Report

“COMPUTATIONAL INTELLIGENCE”

Politecnico di Torino

Academic Year 2023-2024

Filippo Bertolotti 317811

SUMMARY

1. Intro

The aim of the following report is to summarize all the work done while the course of Computational Intelligence hosted by Professor Giovanni Squillero.

All the laboratories and the final project are reported with a description of the solution proposed, the respective key parts of code and the peer reviews.

2. A* Set Covering

2.1 Introduction

The first laboratory involves solving a set cover problem using an **“A* Search Algorithm”**.

Specifically, each element of the universe **U** consists of a certain number **N** of Boolean variables. The objective is to identify the smallest subset of the universe in which performing an OR operation among its members yields a sequence of **N** True values.

2.2 Algorithm

The algorithm follows a strategy of systematic exploration of the solution space by identifying elements with the maximum number of True values in the sets. This heuristic function aims to efficiently guide the search towards optimal solutions with improved computational performance.

The main algorithm initializes by randomly generating a set covering problem, creating a starting state that encapsulates the current selected tiles, those yet to be selected, and the maximum heuristic value.

The iteration process continues until the goal state, where all sets are covered, is achieved. In each iteration, the algorithm prioritizes elements with the maximum heuristic value. It explores neighboring states by selecting or deselecting the identified element based on the heuristic. The state is then updated, and the process repeats.

```python
def goal_check(state, sets):
    return np.all(reduce(
        np.logical_or,
        [sets[i] for i in state.taken],
        np.array([False for _ in range(PROBLEM_SIZE)]),
    ))


def distance(state, sets):
    return PROBLEM_SIZE - sum(
        reduce(
            np.logical_or,
            [sets[i] for i in state.taken],
            np.array([False for _ in range(PROBLEM_SIZE)]),
        ))

def heuristic(i, sets):   # return the tile's number of TRUE values
    h = 0
    for boolean in sets[i]:
        if boolean: h+=1
    return h
```

```python
for _ in range(1) :
    sets = tuple(
        np.array([random() < 0.3 for _ in range(PROBLEM_SIZE)])
        for _ in range(NUM_SETS)
    )

    frontier = SimpleQueue()
    state = State(set(), set(range(NUM_SETS)), PROBLEM_SIZE)
    frontier.put((distance(state, sets), state))
    _, current_state = frontier.get()

    while not goal_check(current_state, sets):
        flag = False
        counter += 1
        for action in current_state[1]:
            H = heuristic(action, sets)
            if H == current_state.MAX_H:    #search starting from the tiles with MAX_H number of TRUE
                new_state = State(
                    current_state.taken ^ {action},
                    current_state.not_taken ^ {action},
                    current_state.MAX_H,
                )
                frontier.put((distance(new_state, sets), new_state))
                flag = True
        if not flag:                        #if "tile not found" decrease MAX_H
            new_state = State(
                current_state.taken,
                current_state.not_taken,
                current_state.MAX_H - 1,
            )
            frontier.put((distance(new_state, sets), new_state))
        _, current_state = frontier.get()

    sol_lenght += len(current_state.taken)
```

2.3 Conclusion

The algorithm is designed to handle a universe consisting of 150 elements, with each element containing 30 Boolean variables. In most cases, the algorithm demonstrates efficiency by identifying a valid solution in fewer than 100 iterations. However, it's noteworthy that there are instances classified as critical cases where the algorithm experiences a decrease in efficiency, requiring more than 1000 iterations to converge and produce a solution.

3. Evolutionary Algorithm & Nim Game

3.1 Introduction

The objective of this laboratory is to implement a virtual player capable of playing the Nim game using an **Evolutionary Algorithm** and make it play against two others virtual players that respectively follow a completely random strategy and an optimal one.

In this implementation the **'individual'** serves as the genome in the evolutionary algorithm. An individual is characterized by a counter, **'wins',** which keeps track of the number of matches won, and a set of parameters, including:

- **'odd'**: the number of rows with an odd number of elements.
- **'first_last'**: the probability to pick elements from the first row.
- **'k'**: the probability to pick all elements from a row.

The initial population consists of 100 individuals with randomly assigned parameters, each ranging from 0 to 1. The counter is initialized to 0 for all individuals.

3.2 Algorithm

The evolutionary strategy is based on a tournament between the 100 individuals. Every individual plays against each other two times, one as first one as second, for a total of 198 matches each. The winner's counter is increased every time a match ends up.

Then the population is sorted in terms of won matches and the worst 50 individuals are discarded while the best 50 individuals are passed to the **'mutation'** function which randomly changes the parameters of every individual and reset the counters.

After that 50 new players are generated, each picking up randomly two parents from the bests individuals and creating a new set of parameters choosing randomly from the parents' ones.

At the end the best individual plays against the Optimal and the Random players 1000 times each, the results are compared with a threshold and if they are not good enough the algorithm restarts.

```python
def mutation(off_spring: None) -> None:
    """mutate randomly individuals' parameters and reset the numbers of won matches"""
    probability = 0.2
    for I in off_spring:
        I._wins = 0
        if random.random() < probability: I._odd = random.random()
        if random.random() < probability: I._k = random.random()
        if random.random() < probability: I._first_last = random.random()
    return off_spring
```

```python
def ES(state: Nim, I: individual) -> Nimply:
    """Evolutionary Strategy"""
    odd_index = count_odd_rows(state)
    odd_ratio = len(odd_index)/ len(state.rows)
    if (odd_ratio > I.oddParam) & (odd_ratio != 0.0):
        """i'm gonna pick from a row with odd number of elements"""
        if random.random() < I.k: N = state.rows[odd_index[0]]  #pick all elements
        else: N = 1     #pick one element
        return Nimply(odd_index[0], N)
    else:
        rowsWithElement = (rows_with_element(state))
        L = len(rowsWithElement)-1
        if random.random() < I.FirstLast:
            """i'm gonna pick from the first row"""
            if random.random() < I.k: return Nimply(rowsWithElement[0], state.rows[rowsWithElement[0]]) #pick all elements
            else: return Nimply(rowsWithElement[0], 1)  #pick one element
        else:
            """i'm gonna pick from the last row"""
            if random.random() < I.k: return Nimply(rowsWithElement[L], state.rows[rowsWithElement[L]]) #pick all elements
            else: return Nimply(rowsWithElement[L], 1)  #pick one element

def ES_match(ES_agent: individual, rival: None) -> int:
    win_count = 0
    for matches in range(1, 1001):
        player = matches % 2
        nim = Nim(5)
        while nim:
            if player == 0: ply = ES(nim, ES_agent)
            else: ply = rival(nim)
            nim.nimming(ply)
            player = 1 - player
        if(player == 0):
            win_count = win_count + 1
    return win_count
```

```python
#create randomly the starter population
off_spring = np.array([individual(random.random(), random.random(), random.random(), 0) for _ in range(LAMB)])

while True:
    LOOP_NUMBER += 1
    for I in off_spring:
        """TOURNAMENT: every individual play two matches against any other individual (one as first to play, one as second)"""
        for matches in range(LAMB - COUNT):
            for first in (0,1):
                """two matches"""
                player = first
                nim = Nim(ROWS)
                agents = (I, off_spring[COUNT+matches])
                while nim:
                    ply = ES(nim, agents[player])
                    nim.nimming(ply)
                    player = 1 - player
                agents[player].addWin()
        COUNT += 1
    off_spring = sorted(off_spring, key=lambda x: x.wins, reverse=True)     #sort the population based on the number of won matches
    off_spring = off_spring[:50]        #take the best 50 players
    off_spring = mutation(off_spring)   #mutate the best 50
    for _ in range(50):
        """generate 50 new individuals from the best 50"""
        parents = (off_spring[random.randint(0, 49)], off_spring[random.randint(0, 49)])    #pick randomly 2 parents from the best 50
        new_I = individual(parents[random.randint(0,1)].oddParam, parents[random.randint(0,1)].FirstLast, parents[random.randint(0,1)].k, 0)
        off_spring.append(new_I)
    COUNT = 1
    """the best player plays against the Random and the Optimal players 1000 matches each"""
    vsRandom = ES_match(off_spring[0], pure_random)
    vsOptimal = ES_match(off_spring[0], optimal)
    if (vsOptimal > 400) & (vsRandom > 650): break      #stop the algorithm if the Best player won a certain number of matches
```

3.3 Conclusion

The evolutionary strategy demonstrates rapid convergence, yielding a player with an average win percentage of:

- 66% against the pure random player.
- 40% against the optimal player.

However these results are commendable, they suggest that the strategy is not yet perfect. There is potential for improvement by introducing additional parameters. With further refinement, the strategy could achieve an average win percentage of at least 75% and 48% against random and optimal players, respectively.

3.4 Reviews

Peer reviewed: Marco Cirone

Date: 22 November 2023

Review:

Hi Marco, i've taken a look to your evolutionary strategy algorithm. First of all i have to say that your code is very clear, especially i like the fact that you used different little functions in order to modulate the project. Maybe the part inside the last big for loop could appears a bit confusing if someone doesn't read it carefully.

I like the way you create and initialize the starter population, your intentions are explicit starting from this point.

An interesting improvement could be trying to train the individuals not only against the "opponent_strategy" but also against others players like the random one or the other individuals.

You have gained a good result with your algorithm, i think that a possible improvement could be adding some more "rules" such as one considering the number of rows with odd elements so as to obtaining more specific probabilities.

Peer reviewed: Salvatore Cabras

Date: 21 November 2023

Review:

Hi Salvatore, you have reached an incredible result with your algorithm.

in my opinion we have to analyze your strategy without the "move_to_win" function beacause you are giving to the evolutionary algorithm an extra help while maybe it must not know such as powerfull move otherwise in certain situations it would be too easy to him win the game.

i think that you could replicate this function with the introduction of some parameters evolving during the evolutionary strategy.

the code is clear but i suggest you to put some more comments, especially if other people are going to look at it, for example i can't understand the range in the for loop, but probably the problem is mine.

4. Solution For BlackBox Problem

4.1 Introduction

The third laboratory requests the implementation of an algorithm capable of solving a problem without any knowledge of its rules (BlackBox Problem).

The primary objective is to achieve a satisfactory solution with minimal reliance on the **'fitness'** function.

In my case, the implemented algorithm is an Evolutionary Algorithm with an initial population of 10 individuals. Each individual consists in a tuple with two parameters:

- A vector of 1000 genomes where each genome could be 1 or 0.
- The fitness value calculated based on the vector itself.

4.2 Algorithm

The algorithm is straightforward. Firstly, the population is initialized by creating ten individuals and sorting them based on their respective fitness value. In each new generation of the population, the worst five individuals (those with the lowest fitness value) are discarded and replaced with an equal number of new individuals.

Each new individual is generated through crossover between two parents chosen from the best five individuals in the population. Subsequently, the result of the crossover undergoes mutation using a Gaussian mutation with a standard deviation of 0.2. Once the new individual is completed, it is added to the population, and the population re-sorted.

Afterwards, the algorithm analyzes the improvement compared to the previous population. If the improvement exceeds 0.1% the variable **'no_improvement'** is set to 0; otherwise, it is incremented by one. When **'no_improvement'** reaches 2000, it indicates that in the last 2000 generations the algorithm hasn't achieved a significant improvement, so it stops and prints the results.

```
INSTANCE = [1,2,5,10]
GENOMES = 1000              #individual size
SIZE = 10                   #population size

def parent_selection(population, I):
    """choose a parent from the best I of the population"""
    parent = population[np.random.randint(0, I)]
    return parent

def xover(parent1, parent2):
    """crossover between the two parents"""
    return [p1 if r < .5 else p2 for p1, p2, r in zip(parent1, parent2, np.random.random(GENOMES))]

def mutate(ind):
    """new individual generated from a gaussian mutation"""
    SD = 0.2
    mutation = np.random.normal(0, SD, GENOMES)
    mutated = [int(round(c + m)) % 2 for c, m in zip(ind, mutation)]    #gaussian mutation
    return mutated
```

```
for I in INSTANCE:
    fitness = lab9_lib.make_problem(I)

    individuals = []
    for k in range(SIZE):
        """initial population"""
        ind = choices([0,1], k=GENOMES)
        fit = fitness(ind)
        individuals.append((ind, fit))

    generations = 1
    no_improvement = 0
    best = 0.0

    individuals = sorted(individuals, key=lambda i:i[1], reverse=True)      #sorting based on fitness

    while True:
        generations += 1
        for k in range(SIZE//2):
            """substitute worst size/2 individuals with new ones"""
            p1, p2 = parent_selection(individuals, SIZE/2), parent_selection(individuals, SIZE/2)
            newInd = mutate(xover(p1[0], p2[0]))
            individuals[SIZE-1-k] = (newInd, fitness(newInd))
        individuals = sorted(individuals, key=lambda i:i[1], reverse=True)  #sorting based on fitness
        if (individuals[0][1] - best > 0):
            if (individuals[0][1] - best)*100 > 0.1:
                no_improvement = 0      #fitness improvement good enough: let's keep going with the algorithm
            best = individuals[0][1]
        no_improvement += 1
        if no_improvement == 2_000: break   #for n times the fitness improvement has not been good enough so stop the algorithm

    FC += fitness.calls     #increment total number of fitness call
```

4.3 Conclusion

The algorithm achieves some good results, in particular, a generic run of the algorithm produced the following data:

- Problem instance 1

Best fitness: 97.10%

Fitness calls: 84305

Number of generations: 16860

- Problem instance 5

Best fitness: 19.34%

Fitness calls: 11900

Number of generations: 2379

- Problem instance 2

Best fitness: 47.55%

Fitness calls: 18050

Number of generations: 3609

- Problem instance 10

Best fitness: 28.98%

Fitness calls: 28450

Number of generations: 568

Total number of fitness calls: 142705

Generally, the best fitness obtained may vary based on the initial population and the random variable of the algorithm.

4.4 Reviews

Peer reviewed: Leonardo Pieraccioli

Date: 5 December 2023

Review:

Hi Leonardo (and Giovanni), i took a look at your code.

First of all i have to say that the code is very clear and the Markdown parts help to understand the algorithm.

I have seen that you got a good results with your evolutionary algorithm, maybe you could add the graphs also for the other instances of the problem in order to have a general view to the results without the need to download the code and try it by myself.

Anyway, i'll try to give you some advice and some way to improve the results:

the crossover between the two parents could be done with bigger slices, maybe also as many slices as the loci are.

the mutation could be implemented as a gaussian mutation.

Finally, i think that the MAX_GENERATIONS limit is too restrictive, in my opinion you could increment it or try to find other conditions to end the algorithm.

Ultimately, i think that you did a good job. keep pushing!

Peer reviewed: Luca Sturaro

Date: 5 December 2023

Review:

Hi Luca, i took a look at your code.

First of all you're code is clear, also thanks to the several comments you add to you're project. You could add a README file in the project folder with some indications about you're work.

Looking at you're results i see that the number of generations for each instance of the problem is good, you have obtained also some good fitness value. I think that the problem is that the number of fitness call is huge for every type of problem. For example, speaking about problem size 10, form the generation 138 and the generation 9309 you get an improvement of only 1.7%. In my opinion you have to stop the algorithm before.

Also try to decrease the population size, i think that doing this the results in terms of fitness value won't change a lot while you'll get a lighter algorithm and a better number of fitness call. Maybe you can try with a population of 20 individuals, they might be enough.

Ultimately, i can say that you did a good job overall. Keep pushing!

5. Reinforcement Learning & Tic-Tac-Toe

5.1 Introduction

The last laboratory requires the implementation of a virtual player that can play Tic-Tac-Toe using a Reinforcement Learning algorithm, starting from zero knowledge about winning strategy. The virtual player is supposed to employ its strategy against a random player.

The algorithm is based on a class called **'History'**. This class contains:

- The state of X player.
- The state of O player.
- A table where each available move is associated with a reward.

The reinforcement learning player, named **'RL_player'**, formulates its strategy using an array of **'History'** elements. Each element describes a different state of the game.

The two players engage in 100.000 training games, during which the RL_player refines its strategy. Subsequently, they play 100 games of "real game", after which the results are obtained.

The random player is a type of **'reinforced random player'.** In the event it has an available winning move, it will choose that specific move to secure a victory in the game. In these states, it doesn't play randomly.

5.2 Algorithm

Initially, the RL_player plays completely randomly but after every move it adds the current state, if not already present, into the history array. In case the current state is already into the history it will choose

from the table related to this state one of the available moves with the highest reward.

After every winning move, both sides, the algorithm will perform an update on the rewards table of the last state of RL_player. There are two possibilities:

- random_player won, means that RL_player's last move was bad and maybe he could block his enemy so that move get a -1 on its reward table.
- RL_player won, means that RL_player's last move was a winning move so that move get a +2 on its reward table.

```
class History:
    """each state of the game has a reward table"""
    def __init__(self, x: set, o: set) -> None:
        available = set(range(1,10)) - x - o
        self._x = set(x)
        self._o = set(o)
        self._tab = {num: Reward(move=num, rew=0) for num in available}
```

```
def find_state_in_history(history, state: (set, set)):
    """check if RL_player is in a state already present in its history"""
    for rew in history:
        if rew._x == state[1] and rew._o == state[0]:
            return rew
    return None

def RL_player(available: set(), state: (set, set), history):
    found = find_state_in_history(history, state)
    if found is not None:
        """it is in a familiar state so choose the best move"""
        max_rew = sorted(found._tab.values(), key=lambda x:x.rew, reverse=True)[0].rew
        possible_move = [move for move, reward in found._tab.items() if reward.rew == max_rew and move in available]
        return random.choice(possible_move)
    else:
        """it isn't in a familiar state so add this state to the history"""
        history.append(History(state[1], state[0]))
        return random.choice(list(available))

def update_rewards(state: (set, set), history, move: int, update: int):
    found = find_state_in_history(history, state)
    if found is not None:
        found._tab[move] = Reward(move=move, rew=found._tab[move].rew + update)
```

```python
# player1: RL_player play with X
# player0: random_player play with O

history = []

def random_game(first: int):
    state = (set(), set())
    available = set(range(1, 10))
    now_playing = first
    while available:
        if now_playing:
            #player1
            RL_play = RL_player(available, state, history)
            previous_state = (set(state[0]), set(state[1]))
            state[1].add(RL_play)
            available.remove(RL_play)
            if win(state[1], RL_play):
                update_rewards(previous_state, history, RL_play, 2)
                return 1
        else:
            #player0
            random_play = random_player(available, state)
            state[0].add(random_play)
            available.remove(random_play)
            if win(state[0], random_play):
                update_rewards(previous_state, history, RL_play, -1)
                return 0
        now_playing = 1 - now_playing
    return '-'
```

```python
# Warm-up
for i in range(100000):
    random_game(first)
    first = 1 - first
    print(i)

# Let's play seriously
for _ in range(100):
    winner = random_game(first)
    first = 1 - first
    if winner == '-': draw += 1
    elif winner == 0: random_player_wins += 1
    elif winner == 1: RL_player_wins += 1
```

5.3 Conclusion

The current strategy might not be optimal for more complex games. However, the algorithm's execution time of two minutes is reasonable. The results against the reinforced random player appear promising, with the RL_player having an average 80% probability of winning after training, a 10% chance of losing, and the remaining portion consisting of games ending in a draw.

5.4 Reviews

Peer reviewed: Gregorio Nicora
Date: 3 January 2024
Review:

Hi Gregorio I took a look to your code.

The code is clear (also thanks to the comments) and well organized.

Sincerely I have not big advices for you because your work is good and the reinforcement player get great results. You can maybe try to implement the RL player with different initial datas in order to see various results.

You can also try to develop the possibility to play against your player in a human VS machine match.

Anyway good job! Keep pushing!

Peer reviewed: Ludovico Fiorio
Date: 3 January 2024
Review:

Hi Ludovico, I took a look to your code.

The code is clear while the README file could be more implemented.

I have seen that you make your reinforcement learning player plays always as first in each match, this aspect could change the final results in favour of your RL player.

Anyway the RL player is well trained especially thanks to the "clever_game" function, I still prefer the random way in order to train the player so it starts with zero knowledge about the game.

One more advice, you could improve the way you show the results with some more informations and also a graph.

Good job! Keep pushing!

6. Quixo

6.1 Introduction

The final project for the course involves implementing a virtual player capable of playing Quixo against another player operating randomly. The choice of the algorithm for the virtual player is flexible.

The chosen algorithm is a Minimax algorithm with depth set to 3. Each time the virtual player is required to make a move, the algorithm explores the decision tree of the current state of the board, searching for a move that could lead to victory and, importantly, avoids moves that would result in losing the game. Specifically, the reward for a move leading to victory is set to 1, -1 for a move leading to defeat, and 0 for a move resulting in a neutral state.

Due to the numerous available moves, the execution might be slow. To enhance efficiency, the virtual player loads a file acting as a cache at the beginning of each match. This file contains previously visited board states with their respective rewards. If the virtual player encounters a known state, the algorithm does not need to visit the decision tree. After the game concludes, the virtual player updates the file with the new discovered states and the corresponding rewards.

The Minimax algorithm naturally designates the random player as the minimizing player and the virtual player as the maximizing one. The random player is associated with the value 0, the virtual player with the number 1, and -1 represents the neutral value.

6.2 Algorithm

The virtual player executes its next move through the **'make_move'** function. Initially, this function retrieves all available moves from the current state (first level of the decision tree). Subsequently, for each move, it calculates its reward by invoking the **'recursive_minimax'** function. This recursive function explores the decision tree, starting from the preceding move and alternating between the maximizing and minimizing player. The recursion halts through two distinct conditions:

- The current state corresponds to a final state, resulting in a return of -1 (defeat) or 1 (victory).
- The **'depth'** parameter passed to the function, decreasing with each recursion, equals zero, leading to a return of 0 (neutral state).

The **'evaluate'** function determines whether the current state should conclude the recursion or not.

If the reward returned is 1, signifying a winning move, the algorithm stops to explore other decision trees and proceeds with that specific move. Otherwise, it continues iterating over the set of available moves, seeking a more favorable game strategy.In the absence of any winning move, the algorithm selects randomly one of the moves leading to a neutral state while avoiding defeat, with the hope that such a move exists.

```
if __name__ == '__main__':
    wins = [0, 0]

    for i in range(100):
        g = Game()
        player1 = RandomPlayer()
        player2 = MyPlayer()
        player2.load_cache("cache")
        wins[g.play(player1, player2)] += 1
        player2.save_cache("cache")
        print(i)
```

```python
def make_move(self, game: 'Game') -> tuple[tuple[int, int], Move]:
    board = Board(game.get_board())
    available_moves = self.get_available_moves(board, 1)
    neutral_moves = []

    """first level of the decision tree"""
    for move in available_moves:
        temp_board = Board(board.get_board())
        temp_board.move(move[0], move[1], 1)
        res = self.recursive_minimax(temp_board, 0, 1)

        if res > 0:
            """winning move found, the search might end, I win"""
            from_pos, slide = move
            return from_pos, slide

        elif res == 0:
            neutral_moves.append(move)

    if len(neutral_moves) > 0:
        """no winning moves, but i can continue to play chosing a neutral move"""
        from_pos, slide = random.choice(tuple(neutral_moves))
    else:
        """there aren't winning or neutral moves, I lose"""
        from_pos, slide = random.choice(available_moves)

    return from_pos, slide
```

```python
def recursive_minimax(self, board: Board, is_maximizing_player: int, deep: int):
    """explore the decision tree"""
    evaluation = self.evaluate(board, deep)
    if evaluation is not None:
        """i can stop the recursion"""
        return evaluation

    available_moves = self.get_available_moves(board, is_maximizing_player)

    evaluation_function = max if is_maximizing_player else min
    evaluation = float('-inf') if is_maximizing_player else float('inf')

    for move in available_moves:
        temp_board = Board(board.get_board())
        temp_board.move(move[0], move[1], is_maximizing_player)
        res = self.recursive_minimax(temp_board, 1 - is_maximizing_player, deep - 1)
        evaluation = evaluation_function(evaluation, res)
        """alpha-beta pruning"""
        if (is_maximizing_player and evaluation > 0) or (not is_maximizing_player and evaluation < 0):
            break

    self.cache[hash(str(board.get_board().tolist()))] = evaluation  # add the state into the cache

    return evaluation
```

```python
def evaluate(self, board: Board, deep: int):
    """check if the recursion might finish or not"""
    winner = board.check_winner()
    if winner == 0:
        """losing move"""
        return -1

    if winner == 1:
        """winning move"""
        return 1

    if deep == 0:
        """last node of the decision tree"""
        return 0

    board_hash = hash(str(board.get_board().tolist()))  # search the reward in the cache
    if board_hash in self.cache:
        """current state already present into the cache"""
        return self.cache[board_hash]

    return None
```

6.3 Conclusion

The algorithm takes approximately two minutes and a half to play 100 games, and the outcomes are quite good. In a test comprising 100 matches, the virtual player employing the Minimax algorithm wins an average of 90 matches.

While increasing the maximal depth could potentially yield better results, the trade-off is a significant increase in execution time.

The cache's efficiency is notably impacted by the extensive number of different combinations – approximately 850 billion – of the board states. To address the challenge posed by the multitude of combinations, an approach involves analyzing the symmetry of the board. By applying transformations such as rotations or inversions, certain states become equivalent, reducing the number of distinct combinations to around 100 billion.