Artificial Intelligence 1
Lab 2

Oskar Gryglewski (s5133122)
Mihkel Mariusz Jezierski (s4787730)
Aksel Joonas Reedi (s4790820)
Learning Group 6

May 12, 2023

Disclosure

Collaboration on the code by: Oskar Gryglewski (s5133122); Mihkel Mariusz Jezierski (s4787730); Aksel Joonas Reedi (s4790820)

To develop the code we used the following resources: 1) Learning materials provided by the course 2) Russell, S., Norvig, P. (2014). Artificial Intelligence: a Modern Approach (Third edition, Pearson new international). Pearson.

1 N-queens problem

1.1 Hill Climbing

a)

The Hill Climbing algorithm successfully finds solutions for N = {4, 8}, more often than not does not find solution for N = 16, and is completely unsuccessful for N = {32, 64}.

b)

Our implementation of the Hill Climbing algorithm for the given N Queens problem fails to find a solution for larger state sets than N = 16. The main reason is in the algorithm's fundamental design itself. Since it works on the basis of picking the best neighbour state without any other heuristic evaluation, it is prone to get stuck on local maximum states. With a larger number of queens, there are inherently more states, therefore, more local maximum states for which the conventional Hill Climbing algorithm does not have any functionality to handle.

c)

There are numerous ways how to improve the Hill Climbing algorithm. One would be to deal with state shoulders (where there are no better states, only equal ones). Secondly, it would be to implement some randomness when the local maximum is reached. Furthermore, currently, our code changes one queen's position (not excluding the same position) and immediately evaluates it, but this could be minimised when the random position is generated until the position is different. We decided to focus on the latter idea. Hence, in addition to the neighbour search limit (100) and the randomness (which is used when the neighbour state's evaluation is equal to the current state), which were both implemented since our traditional Hill Climbing was heavily underperforming, we changed how the neighbours are created. Instead of creating a completely random neighbour and evaluating it, we decided to separate the column and the row. Firstly, the column is picked and then the row to which the queen is placed. The new state is evaluated only if it is not the same state as before.

first Hill Climbing implementation (function for picking the best neighbour)

```
def find_best_neighbor(board):
 best_board = board.copy()
    new_board = board.copy()
    for i in range(100):
        new_board[random.randint(0, len(board)-1)] = random.randint(0,
            len(board)-1)
        if evaluate_state(new_board) > evaluate_state(best_board):
               best_board = new_board.copy()
        elif evaluate_state(new_board) == evaluate_state(best_board):
               if random.randint(0, 1) == 1:
                   best_board = new_board.copy()
    return best_board
```

optimised/improved Hill Climbing implementation (function for picking the best neighbour)

```
def find_best_neighbor(board):
    best_board = board.copy()
    for i in range(100):
        row = random.randint(0, len(board)-1)
        column = random.randint(0, len(board)-1)
        while board[column] == row:
            row = random.randint(0, len(board)-1)
        new_board = board.copy()
        new_board[column] = row
        if evaluate_state(new_board) > evaluate_state(best_board):
            best_board = new_board.copy()
```

```
        elif evaluate_state(new_board) == evaluate_state(best_board):
            if random.randint(0, 1) == 1:
                best_board = new_board.copy()
    return best_board
```

d)

N = number of queens	unoptimised Hill Climbing (out of 10 tries)	improved Hill Climbing (out of 10 tries)
N = 4	10	10
N = 8	10	10
N = 16	4	10
N = 32	0	10
N = 64	0	9

We observe that the improvement made the search algorithm considerably better. In fact, the improved Hill Climbing algorithm was even capable of solving the N Queens problem with N = 100. Though, it should be noted that even the unoptimised version of our Hill Climbing implementation has some elements (e.g., only 100 neighbours are evaluated instead of all neighbours, which makes the algorithm considerably faster. This was a suggestion by the teaching assistant) that are missing from the original pseudo-code presented in the book.

1.2 Stimulated Annealing

a)

Our implementation of simulated annealing is based on the pseudo-code provided in the course book. It is a logical combination of random search and a hill-climbing algorithm. It takes an initial state as an input and in each iteration, it generates one random neighbouring state. If the new state is better (this is determined by the delta$_e variable, which is a subtraction between the number of conflicting queens both states have) than advancing states$.

b)

We used the following formula to calculate the temperature T:

```
def time_to_temperature(time):
    return 1000 * pow(0.999, time)
```

The starting temperature was set to 1000. With each iteration (indicated by the time in the function), it decreased at the rate of 1 per cent. We chose this formula based on trial and error because we noticed that the linear implementation (where with each iteration, the temperature decreased by one) did not produce reasonable results.

c)

N = number of queens	simulated annealing (out of 10 tries)
N = 4	10
N = 8	10
N = 16	10
N = 32	10
N = 64	10

d)

When we decreased the rate at which the temperature decreased, the performance dropped because the algorithm was more likely to pick choices that did not minimise the total number of conflicting queens. Therefore it was more prone to make negative moves and run out of the iteration limit. On the other hand, when the temperature decrease rate was made bigger, the randomness effect was minimised and the algorithm was more prone to get stuck at local minima before the temperature hit 0 (which also happened faster than with the regular decrease rate).

1.3 Genetic Algortihm

a)

We followed the pseudo-code in the book to get our first implementation of the genetic algorithm. It takes in an initial board and generates a random population of boards according to the set value of `POPULATION_SIZE`. The population size is set to 150, which seems to be a reasonable balance between diversity and efficiency.

```
1 def genetic_algortihm(board):
2     iterations = 0
3     optimum = (len(board) - 1) * len(board) / 2
4 
5     population = []
6     population_fitness = []
7     POPULATION_SIZE = 150
8     NEW_GEN_CUT = 0.2
```

```
    MUTATION_PROBABILITY = 0.1

    # generate 1st population and their fitnesses
    for i in range(POPULATION_SIZE):
        temp = init_board(len(board))
        population.append(temp)
        population_fitness.append(get_fitness(temp))
```

Children are generated in a way, where a random 'cut' point is chosen, and the child inherits genes from the first parent up to the cut point and from the second parent from the cut point.

```
def reproduce(x,y):
    cut = random.randint(0, len(x)-1)
    return x[:cut] + y[cut:]
```

The mutation is designed to introduce diversity into the population. It occurs with a probability of 0.1 (`MUTATION_PROBABILITY`) and changes a random position in the child to a random number.

```
def mutate(child):
    child[random.randint(0, len(child)-1)] = random.randint(0,
        len(child)-1)
    return child
```

After each generation, the top 20 of the population (by the `NEW_GEN_CUT` parameter) is used to generate the entirety of the next generation. This selection is simple and effective. However, this approach sometimes leads to getting stuck in local extremes and does not find a solution. We also tried to solve it by using "Stochastic Universal Sampling (SUS), but we did not get it working in time for the deadline.

The algorithm stops when the optimal solution is found, or when the maximum number of iterations (1000) is reached.

```


    while evaluate_state(board) != optimum:
        iterations += 1
        print('iteration ' + str(iterations) + ': evaluation = ' +
            str(evaluate_state(board)))
        if iterations == 1000: # Give up after 1000 tries.
            break

        #map fitness to state and sort the lists
        reproduction, reproduction_fitness = [list(a) for a in
            zip(*sorted(zip(population, population_fitness), key=lambda
            pair: pair[1], reverse=True))]

        new_population = []
```

```
        new_population_fitness = []

        for i in range(POPULATION_SIZE):

            # select best for reproduction
            mom = random.choice(reproduction[:int(POPULATION_SIZE *
                NEW_GEN_CUT)])
            dad = random.choice(reproduction[:int(POPULATION_SIZE *
                NEW_GEN_CUT)])

            child = reproduce(mom, dad)

            if (random.random() < MUTATION_PROBABILITY):
                child = mutate(child)

            new_population.append(child)
            new_population_fitness.append(get_fitness(child))

        board = reproduction[0]
```

b)

Which of the three methods (Hill climbing, simulated annealing, and genetic algorithms) works best for the N-queens problem (for N= 4,8,16, averaging over ten runs) and why does that algorithm outperform the rest?

N = number of queens	Hill-climbing	Simulated annealing	Genetic algorithm
N = 4	10	10	10
N = 8	10	10	10
N = 16	10	10	10
N = 32	10	10	10
N = 64	9	10	10

2 Programming assignment - Game of Nim

Questions

a)

- n=3: In the first move, MAX takes 2 matches. As a result, only 1 match is left, which MIN must take and therefore loses.

  Looking at MAX's initial moves:

- If MAX takes one match, MIN can choose to take one or two matches. If MIN picks two, it loses and MAX wins, resulting in a utility value of 1 for MAX. But taking one match would leave one match for MAX, causing MAX's loss and a utility value of -1.
- If MAX takes two matches, only one is left for MIN to take, leading to MIN's defeat and a utility value of 1 for MAX.
- If MAX takes all three matches, it results in MAX's instant loss and a utility value of -1.

Therefore, the optimal move for MAX is to take two matches for an immediate win and a utility value of 1.

- n = 4: MAX takes 3 matches initially. Only 1 match remains, which MIN must take and therefore loses.

  Looking at MAX's initial moves:

  - If MAX takes one match, there are three matches left. This situation is similar to the previously discussed scenarios, but with inverted utility values as now we're considering a MIN sub-tree. So, taking one match in the beginning results in a utility value of -1 for MAX.
  - If MAX takes two matches, it leads to a MIN sub-tree with 2 as a root, which results in a utility value of -1 for MAX.
  - If MAX takes three matches, only one match is left for MIN to take, leading to MIN's loss and a utility value of +1 for MAX.

  In this case, MAX's only winning move is to initially take three matches.

- n = 5: MAX takes 1 match initially. MIN then takes 3 matches. Only 1 match remains, which MAX must take and therefore loses.

  Looking at MAX's initial moves:

  - All scenarios, whether MAX takes one, two, or three matches, lead to a utility value of -1. In these situations, the states can be represented as MIN sub-trees, and all utility values are swapped. Hence, no matter the initial move, MAX can't win in this case.

- n = 6: MAX takes 1 match initially. MIN then takes 1 match. MAX then takes 3 matches. Only 1 match remains, which MIN must take and therefore loses.

  Looking at MAX's initial moves:

  - If MAX takes one match, it leads to a MIN sub-tree with root value 5, resulting in a utility value of -1 for MAX.
  - If MAX takes two matches, the situation is the same as the case where n=5 and one match was taken initially, leading to a utility value of -1.

– If MAX takes three matches, the situation is the same as the case where n=4 and one match was taken initially, leading to a utility value of +1.

In this case, MAX can win by making the initial move of taking one match.

b)

In each of the tests (n = 10, n = 20, n = 30) MAX wins the game and the move k = 2 is never used. For any n where n % 4 != 0, MAX can always force a win by making the number of matches left a multiple of 4 after their turn. When n % 4 == 0, MIN can adopt this strategy and force a win. However, one noticeable difference between the tests is the run time. For n = 10 and n = 20, it's relatively short, but for n = 30 it's significantly longer. That is because the search tree grows exponentially with increasing n.

c)

We implemented a transposition table in the form of the `t_table` variable. This table stores the utility values of each state for both MAX and MIN. It serves to avoid redundant calculations and speed up the decision-making process.:

```
1  def play_nim(state):
2      t_table = [[None for x in range(2)] for y in range(state+1)] //
           added line
3      turn = 0
4      while state != 1:
5          move, valuation = negamax_decision(state, turn, t_table)
6
7          print(str(state) + ": " + ("MAX" if not turn else "MIN") + "
               takes " + str(move) + " valuation " + str(valuation))
8
9          state -= move
10         turn = 1 - turn
11
12     print("1: " + ("MAX" if not turn else "MIN") + " looses")
13 def max_value(state, t_table):
14     max = -100000000000
15
16     if state == 1:
17         return -1
18
19     for move in range(1, 4):
20         if state-move > 0:
21             if t_table[state-move][1] == -1:
22                 m = -1
23             elif t_table[state-move][1] == 1:
24                 m = 1
25             elif t_table[state-move][1] == None:
```

```
                m = min_value(state-move, t_table)
                t_table[state-move][1] = m
            max = m if m > max else max

    return max


def min_value(state, t_table):
    min = 10000000000000

    if state == 1:
        return 1

    for move in range(1, 4):
        if state-move > 0:
            if t_table[state-move][0] == -1:
                m = -1
            elif t_table[state-move][0] == 1:
                m = 1
            elif t_table[state-move][0] == None:
                m = max_value(state-move, t_table)
                t_table[state-move][0] = m
            min = m if m < min else min

    return min
```

d)

The transposition table would help for n = 50. The reason is that the minimax algorithm has to compute utility values for a number of game states which can be quite large for bigger games (like n = 50). By using a transposition table, we can store these computed values and look them up when needed, instead of recomputing them. This can significantly speed up the computation.