UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

LAB 1: N-Queens Problem

SUBJECT: Artificial Intelligence Fundamentals

Class: 21CLC03

Student: Trần Nguyên Huân

Student ID: 21127050

Instructors: Nguyễn Ngọc Thảo, Lê Ngọc Thành,

Nguyễn Trần Duy Minh, Nguyễn Hải Đăng

HO CHI MINH CITY – 2023

Contents

I. Check list

Tasks	**Done or not**
Presentation detail about 3 algorithms (explain each function) and implement those search algorithms in Python.	Done
Presentation a brief description of main function	Done
Measuring their running times and consumed memory	Done
Giving observation and comments on the statistic	Done
Giving the subjective evaluation based on statistics	Done
Report and source	Done

II. Description each function in algorithms and main function (UCS, A*, Genetic Algorithm)

II.1. UCS

- The **Node** class represents a node in the search tree. It has two attributes: **state** and **cost**. The **state** attribute is a list representing the state of the board. Each element of the list represents the row number of the queen in the corresponding column. The **cost** attribute represents the cost of the node, which is the number of pairs of queens that are attacking each other.

- The **count_conflicts** function takes a state as input and returns the number of pairs of queens that are attacking each other.

- The **uniform_cost_search** function takes an integer **n** as input, which represents the size of the board. It also takes an optional integer **max_steps** as input, which represents the maximum number of steps the algorithm is allowed to take before giving up. The function returns a solution to the N-Queens problem if one is found, or **None** otherwise.

+ The function starts by initializing a random state and its corresponding cost as the starting node. It then initializes an empty priority queue and adds the starting node to it. The **heapq** module is used to maintain the priority queue, with the lowest cost node having the highest priority.

+ The function then enters a loop that continues until the priority queue is empty or the maximum number of steps is reached. In each iteration of the loop, it pops the node with the lowest cost from the priority queue and checks if it is a goal state (i.e., if its cost is zero). If it is a goal state, the function returns the state. Otherwise, it generates the successor states by considering all possible moves of a single queen and adds them to the priority queue with their corresponding costs. Finally, it increments the step counter and continues the loop.

+ If the loop exits without finding a solution, the function returns **None**.

II.2. Graph search A* with MIN-CONFLICT heuristic

- get_successors(node): This function takes a node as input and returns a list of its successors with their respective costs. Each successor is generated by changing the position of one queen to an unoccupied position in the same row. The cost of a successor is calculated as the sum of the number of conflicts in the board state after the queen is moved and the heuristic cost of the resulting state.

- The **count_conflicts** function takes a state as input and returns the number of pairs of queens that are attacking each other.

- heuristic(node): This function takes a node as input and returns its heuristic cost, which is the number of conflicts in the board state.

- is_goal(node): This function takes a node as input and returns **True** if it represents a goal state, i.e., a board state with no conflicts.

- a_star_search(n): This is the main function that takes an integer **n** as input and returns the solution to the n-queens problem using the A* search algorithm. It first initializes a random board state and then uses the **get_successors()**, **heuristic()**, and **is_goal()** functions to generate and evaluate the states until a goal state is reached.

+ The algorithm maintains a priority queue called **frontier** to store the nodes yet to be explored. Each node in the queue is associated with a cost, which is the sum of the heuristic cost and the cost to reach that node. The algorithm uses the priority queue to always select the node with the lowest cost for expansion. The nodes that have already been explored are stored in a set called **explored.**

+ The algorithm keeps exploring the nodes until it reaches a goal state or the priority queue becomes empty, which indicates that there is no solution to the n-queens problem with the given board size. If a goal state is reached, the algorithm returns the board state as a list of integers. Otherwise, it returns **None**.

II.3. Genetic Algorithm with MIN-CONFLICT heuristic

- generate_population(n, N): This function generates an initial population of N random solutions, each representing a possible configuration of n queens on a chessboard. It takes two arguments: n, which is the number of queens, and N, which is the size of the population to be generated. It returns a list of N lists, where each inner list is a permutation of the integers from 0 to n-1.

- fitness(solution): This function computes the fitness of an n-queens solution. It takes a list of integers as input, where each integer represents the row position of a queen on a chessboard. The function returns a score that is higher for better solutions (i.e., those with fewer conflicts). The fitness score is computed by counting the number of conflicts between pairs of queens, where a conflict occurs if two queens share the same row, column, or diagonal. (Similar to count_conflict but fitness function get idea that The higher the number of conflicts, the lower the fitness score).

- tournament_selection(population, M, k): The function uses a for loop to repeat the tournament process **M** times, selecting one individual from each tournament to form the new population. In each tournament, **k** individuals are randomly selected from the original population, and the individual with the highest fitness score (determined by the fitness function) is chosen as the winner. The winner of each tournament is added to the **selected** list. Finally, the function returns the **selected** list of **M** individuals, which represents the new population that will be used in the next generation of the genetic algorithm.

- crossover(parent1, parent2): This function generates a child solution by applying one-point crossover to two parent solutions. It takes two lists of integers as input, where each integer represents the row position of a queen on a chessboard. The function randomly selects a crossover point, and then combines the first part of parent1 with the second part of parent2 to create the child solution.

- **mutation(solution)**: This function mutates a solution by randomly swapping two queens. It takes a list of integers as input, where each integer represents the row position of a queen on a chessboard. The function randomly selects two positions in the solution and swaps the values at those positions.

- **genetic_algorithm(n, N, M, k, max_steps)**: This function solves the n-queens problem using a genetic algorithm. It takes five arguments: n, which is the number of queens; N, which is the size of the population; M, which is the number of individuals to select in each generation; k, which is the number of competitors in each tournament; and max_steps, which is the maximum number of generations to run the algorithm. The function returns the best solution found by the algorithm, or None if no solution is found within the maximum number of generations. The algorithm iteratively applies selection, crossover, and mutation to generate new generations of solutions, and terminates when a solution with zero conflicts is found or the maximum number of generations is reached.

II.4. Main Function

- The main is designed allows a user to input the size of the board. It then prompts the user to choose one of three algorithms to solve the problem: Uniform-cost search, A* with MIN-CONFLICT, or Genetic algorithm.

- The **run_algorithms()** function is then called with the input size and algorithm choice as parameters, and the resulting solution (if any) is stored in the **solution** variable.

- This program measures the time and memory usage of the program by using the **time** and **tracemalloc** modules. **tracemalloc.start()** and **tracemalloc.stop()** functions are used to start and stop tracing memory usage. The memory usage before running the algorithm is stored in the **mem_usage_before** variable and the memory usage after running the algorithm is stored in the **mem_usage_after** variable. The difference between these two variables gives the memory usage of the program.

- Finally, the program prints the solution (if any) using the **print_board()** function, the elapsed time of the algorithm execution, and the memory usage of the program.

III. Performance of those algorithms (running times and consumed memory)

III.1. Measured results table

	Running time (ms)			Memory (MB)		
Algorithms	N = 8	N = 100	N = 500	N = 8	N = 100	N = 500
UCS	154.383	Intractable	Intractable	0,0044	Intractable	Intractable
A*	31.2267	2272433.48	Intractable	0.1328	0.1119	Intractable
GA	31.967	2775933.65	Intractable	0.0047	0.3634	Intractable

III.2. Observations and comment on the statistics

* Observation:

- For N = 8, all three algorithms were able to find a solution within a reasonable amount of time (less than 1 second) and memory usage (less than 0.2 MB).
- UCS algorithm was able to solve the N=8 problem within a reasonable time and with low memory usage, but it became intractable for larger problems (N=100 and N=500) due to its exponential running time.
- The A* algorithm and GA were able to find a solution for N = 100, but took significantly longer and used more memory than for N = 8. For N = 500, both algorithms were intractable and could not find a solution within a reasonable amount of time or memory usage. A* algorithm used more memory than UCS algorithm, but the memory usage was still reasonable for small to medium-sized problems.
- The A* algorithm and GA required significantly more memory than the UCS algorithm for N = 100, but had faster execution times. However, for N = 500, the memory usage for both algorithms was intractable, and the execution times were also very high.

* Comments:

- ➢ The UCS algorithm is not well-suited for large board sizes due to its high execution time, and becomes intractable beyond a certain size.
- ➢ The A* algorithm and GA are able to solve the N-Queens problem for larger board sizes, but require significantly more memory usage and execution time.
- ➢ The memory usage of the algorithms is also an important consideration, especially for problems with large search spaces. UCS algorithm used the least amount of memory, followed by A* algorithm and GA algorithm.
- ➢ The time and memory usage results may vary depending on the implementation of the algorithms used.

- ➢ The intractability of the algorithms for larger problems highlights the need for more efficient algorithms or problem-specific optimizations to tackle the N-queens problem at larger scales.

* Subjective evaluation:

- UCS is a brute-force algorithm that explores all possible paths until a solution is found. It has a low memory usage, but the running time grows exponentially as the board size increases, making it intractable for larger board sizes. The UCS algorithm was unable to find a solution for N = 100 and N = 500 within a reasonable amount of time or memory usage.
- A* with MIN-CONFLICT is an informed search algorithm that uses heuristics to guide the search towards the goal state. It has a higher memory usage compared to UCS but can find solutions faster, particularly for smaller board sizes. However, for larger board sizes, the execution time and memory usage increase exponentially, and it was unable to find a solution for N = 500 within a reasonable amount of time or memory usage.
- The running time of GA algorithm depends on the number of generations and the fitness function used. In general, GA algorithm can converge to a near-optimal solution in a reasonable amount of time, but it may not always find the optimal solution. So UCS and A* algorithms are suitable for solving search problems, while GA algorithm is suitable for solving optimization problems.

IV. References

- Slide lecture:
https://drive.google.com/drive/folders/1NbfqXCKqZf3ECez2oAvEBmJ2rxqo82fx

- Measure memory usage in python (module tracemalloc):
https://www.geeksforgeeks.org/monitoring-memory-usage-of-a-running-python-program/

- Module tracemalloc: https://docs.python.org/3/library/tracemalloc.html

- Measure running time in python (module time): https://www.geeksforgeeks.org/how-to-check-the-execution-time-of-python-script/

- Module time: https://docs.python.org/3/library/time.html

- Module random: https://www.w3schools.com/python/module_random.asp- Module heapq: https://docs.python.org/3/library/heapq.html