VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE

ARTIFICIAL INTELLIGENCE

REPORT

Lab01: N-queens problem

21127229 – Dương Trường Bình

Teacher

Nguyễn Ngọc Thảo

Lê Ngọc Thành

Hồ Thị Thanh Tuyến

Nguyễn Trần Duy Minh

Table of Contents

1. Problem Description 3
2. Search Algorithms 3
3. Implementation 3
   3.1. Program Structure 3
   3.2. Program Flow 7
   3.3. Main Functions 8
4. Performance of Search Algorithms 17
   4.1. Computer Configuration 17
   4.2. Runtime and Memory Measurement 17
   4.3. Measurement Results 18
   4.4. Explanation 18
5. References 23

1. Problem Description

The N-queens problem is a puzzle where the goal is to place N queens on an NxN chessboard such that no two queens threaten each other. In this problem, the queens can attack each other if they are on the same row, same column, or same diagonal.

- **Formulation**: In this lab, the problem is represented using a complete-state formulation, where a configuration represents the placement of all N queens on the board. Each queen is placed in a separate column to ensure that no two queens are in the same column.
- **Successors**: The successors of any state in the problem are all possible configurations generated by moving a single queen to another square in the same column. The objective is to find a configuration where no two queens attack each other, which means finding a solution where all N queens are placed on the board without any conflicts
- **Objective**: The N-queens problem aims to find a configuration where no two queens threaten each other, essentially a solution where all N queens are placed on the chessboard without any conflicts.

2. Search Algorithms

Three search algorithms are implemented to solve the above problem:

- Uniform Cost Search with path cost 1,
- Graph-search A* with min-conflict heuristic: “the number of attacking pairs of queens”,
- Genetic algorithm with the objective function defined from the above heuristic.

3. Implementation

3.1. Program Structure

The program to solve the N-Queens problem using the three search algorithms above is structured following Object-Oriented Programming (OOP) principles.

- **Class Program**:
  - Description: A main program to run and interact with search algorithms.
  - Attributes:
    - *problem*: An instance of the NqueensProblem class

        - *result*: The result of the search algorithm, includes the goal state and the amount of memory used
    - Methods:
        - *run()*: Executes the program by accepting user input for the number of queens and the algorithm choice. The method runs the selected search algorithm on the N-Queens problem instance and produces the following output.
- **Class NqueensProblem**
    - Description: Represents the N-Queens problem and its specific configurations
    - Attributes:
        - *n*: An integer represents the number of queens.
        - *initial_state*: The initial state of the board.
    - Methods:
        - *random_state()*: Generates a random state for the board.
        - *actions(state)*: Generates possible actions or moves for a given state.
        - *goal_test(state)*: Checks if the given state is a goal state.
        - *heuristic(state)*: Calculates the heuristic cost of the given state.
        - *heuristic_change(state, queen_index, old_pos, new_pos, h)*: Updates the heuristic cost when changing the position of a queen.
        - *print_state(state)*: Prints the state of the board.
- **Class Search** (abstract class)
    - Description: Represents a search algorithm
    - Methods:
        - *run()*: Abstract method to run the search algorithm
- **Class Ucs** (inherit from Search)
    - Description: Implements the Uniform Cost Search algorithm.
    - Attributes:
        - *frontier*: Utilizes Python's heapq module to implement a priority queue, enabling efficient storage and retrieval of nodes to be expanded.
        - *explored*: A set to store explored nodes.
    - Methods:
        - *run(problem)*: Executes the Uniform Cost Search algorithm.

        - *get_memory_usage()*: Calculates the memory usage of the search algorithm.
        - *childNode(parent, action)*: Generates a child node from a parent node and an action.
- **Class Astar** (inherit from Search)
    - Description: Implements the A* algorithm.
    - Attributes:
        - *frontier* Utilizes Python's heapq module to implement a priority queue, enabling efficient storage and retrieval of nodes to be expanded.
        - *explored*: Set to store explored nodes.
    - Methods:
        - *run(problem)*: Executes the A* algorithm.
        - *childNode(parent, action, problem)*: Generates a child node from a parent node and an action.
        - *get_memory_usage()*: Calculates the memory usage of the search algorithm.
- **Class GeneticSearch** (inherit from Search)
    - Description: Implements the Genetic Search algorithm.
    - Attributes:
        - *population*: A list to store the population of nodes.
    - Methods:
        - *run(problem)*: Executes the Genetic Search algorithm.
        - *reproduce(parent1, parent2, problem)*: Generates a child node by reproducing two parent nodes.
        - *select_parents(population)*: Selects parents for reproduction based on their fitness.
        - *mutate(node, problem)*: Applies mutation to a node.
        - *get_memory_usage()*: Calculates the memory usage of the search algorithm.
- **Class Node**
    - Description: Represents a node in the search algorithm with a state.
    - Attributes:

        - *state*: State of the node is a list containing integers representing the positions of the queens in each column. Each element in the list corresponds to the row index of a queen in its respective column.

- **Class NodeUcs** (inherits from Node)
    - Description: Represents a node in the Uniform Cost Search algorithm.
    - Attributes:
        - *state*: State of the node is a list containing integers representing the positions of the queens in each column. Each element in the list corresponds to the row index of a queen in its respective column.
        - *path_cost*: The cost of the path to reach this node.
- **Class NodeAstar** (inherits from Node)
    - Description: Represents a node in the A* algorithm.
    - Attributes:
        - *state*: State of the node is a list containing integers representing the positions of the queens in each column. Each element in the list corresponds to the row index of a queen in its respective column.
        - *path_cost*: The cost of the path to reach this node.
        - *heuristic_cost*: The heuristic cost of the node, calculated as the number of queens attacking each other on the board.
- **Class NodeGenetic** (inherits from Node)
    - Description: Represents a node in the Genetic Search algorithm.
    - Attributes:
        - *state*: State of the node is a list containing integers representing the positions of the queens in each column. Each element in the list corresponds to the row index of a queen in its respective column.
        - *heuristic_cost*: The heuristic cost of the node, calculated as the number of queens attacking each other on the board.

This program utilizes several libraries and modules to implement various functionalities. Here's a description of the libraries/modules used in the program:

- *random*: This library is used to generate random numbers for initializing the state of the N-Queens problem, selecting positions, and parents during the Genetic Search algorithm...
- *abc* (Abstract Base Classes): The abc module is imported to define an abstract class (Search) and abstract methods (run) that are inherited and implemented by the search algorithm classes.
- *heapq*: This module is used to implement a priority queue in the Uniform Cost Search (UCS) and A* algorithms. It provides efficient insertion and retrieval of nodes based on their priority (path cost or path cost + heuristic cost) and is crucial for frontier management in these search algorithms.
- *sys*: The sys module is used to calculate the memory usage in the search algorithms. It provides functions to measure the size of objects and data structures, which helps estimate the memory consumed by the frontier, explored sets, and population.

3.2. Program Flow

1. The program starts by creating an instance of the Program class, which serves as the entry point for the program.
2. The user is requested to input the number of queens and the algorithm choice (UCS, A*, or Genetic Search).
3. Based on the user's input, the program instantiates the appropriate search algorithm class (Ucs, Astar, or GeneticSearch).
4. The NQueensProblem class is instantiated, taking the number of queens as a parameter. The initial state of the board is randomly generated.
5. The program executes the chosen search algorithm by calling its run() method, passing the NQueensProblem instance as an argument.
6. The search algorithm performs the search process according to its specific logic:
   a. For UCS and A*, the algorithm initializes the frontier (priority queue) and explored set. It expands nodes, checks for goal state, and continues until a solution is found or the frontier is empty.
      1. For Genetic Search, the algorithm starts with a population of nodes representing potential solutions. It selects parents, reproduces and mutates

offspring, and evolves the population until a solution is found or the number of generations exceeds 100000

7. During the search process, the program may output additional information such as the state of nodes being explored and the minimum heuristic cost.
8. Once the search algorithm completes, the program displays the solution state or reports if no solution is found. It also provides memory usage information specific to each algorithm.

3.3. Main Functions

1. **Program.run()**
    - Input: None
    - Output: None
    - Description: This function serves as the entry point of the program. It requests the user to input the number of queens and the algorithm choice. It then instantiates the appropriate search algorithm class and executes the selected algorithm.
2. **NQueensProblem.init(n)**
    - Input: *n* - The number of queens.
    - Output: None
    - Description: Initializes an instance of the NQueensProblem class by setting the number of queens and generating a random initial state for the board.
3. **NQueensProblem.random_state()**
    - Input: None
    - Output: A list representing a random initial state of the board.
    - Description: Generates a random initial state for the N-Queens problem. Each element in the list represents the row index of a queen in a column.
4. **NQueensProblem.actions(state)**
    - Input: *state* - The current state of the board.
    - Output: A generator that yields possible actions or moves for the given state.
    - Description: Generates all possible actions or moves that can be applied to the current state. Each action represents moving a queen to a different row within the same column.

5. **NQueensProblem.goal_test(state)**
    - Input: *state* - The current state of the board.
    - Output: Boolean (True or False)
    - Description: Checks if the given state is a goal state, where no two queens threaten each other.
6. **NQueensProblem.heuristic(state)**
    - Input: *state* - The current state of the board.
    - Output: The heuristic cost of the state.
    - Description: Calculates the heuristic cost of the given state, which represents the number of conflicts or threats between queens.
7. **NQueensProblem.heuristic_change(state, queen_index, old_pos, new_pos, h)**
    - Input:
        - *state* - The current state of the board.
        - queen_index - The index of the queen being moved.
        - old_pos - The previous position of the queen.
        - new_pos - The new position of the queen.
        - h - The current heuristic cost of the state.
    - Output: The updated heuristic cost of the state after moving the queen.
    - Description: Updates the heuristic cost of the state by considering the change in conflicts caused by moving a queen from old_pos to new_pos.
8. **NqueensProblem.print_state(state)**
    - Input: *state* - The state of the board.
    - Output: None
    - Description: Prints the state of the board, representing the positions of the queens. Each row is printed as a separate line, where a "Q" denotes the presence of a queen in that row and column, and an asterisk (*) represents an empty cell.
9. **Search.run(problem)**
    - Input: *problem* - An instance of the NQueensProblem class.
    - Output: Varies based on the search algorithm.

- Description: Abstract method representing the main execution of a search algorithm. This method is implemented differently in each search algorithm class.

10. Ucs.run(problem)

- Input: *problem* - An instance of the NQueensProblem class.
- Output: Tuple containing the solution node, frontier size, and explored size.
- Description: Executes the Uniform Cost Search (UCS) algorithm to find a solution for the N-Queens problem. It expands nodes based on their path cost and maintains a priority queue (frontier) for efficient exploration.
- Procedure:
  1. Initialize the frontier and explored set.
  2. Create a NodeUcs instance with the initial state and path cost of 0, and push it onto the frontier using heapq.heappush with the path cost as the priority.
  3. Enter a loop until a goal state is found or the frontier becomes empty.
  4. If the frontier is empty, return None to indicate that no solution was found.
  5. Pop the node with the lowest path cost from the frontier.
  6. Print the node's state to visualize the search process.
  7. Check if the node's state satisfies the goal condition. If so, return the node and memory usage.
  8. Add the node's state to the explored set.
  9. Generate child nodes by applying actions to the current node's state.
  10. For each child node:
      - If its state is not in the explored set and not in the frontier, push it onto the frontier.
      - Else if it is in the frontier but has a lower path cost, update the corresponding node in the frontier.

11. Ucs.childNode(parent,action)

- Input:
  - *parent* - The parent node from which the child node is derived.
  - *action* - The action representing the movement of a queen to a new row in the same column.
- Output: A new instance of the NodeUcs class representing the child node.

- Description: This function generates a child node from a parent node copying the parent's state and updating the row of a queen based on the action. The path cost of the child node is also updated by incrementing the parent's path cost by 1. The resulting child node is returned as the output.
- Procedure:
    1. Create a copy of the parent node's state to modify.
    2. Apply the action to the child state by updating the queen's position in the corresponding column.
    3. Create a new NodeUcs instance for the child node with the updated state and a path cost incremented by 1.
    4. Return the child node.

12. Ucs.get_memory_usage()

- Input: None
- Output: Tuple containing the frontier size and explored size in megabytes (MB).
- Description: Calculates the memory usage of the Uniform Cost Search (UCS) algorithm. The function estimates the memory occupied by the frontier (priority queue) and the explored set.
- Procedure:
    1. Get the length of the frontier list and the number of queens (n_queens).
    2. Because the frontier store item in structure: (cost, node) so the memory size (frontier_size) can be calculated by multiplying the length of the frontier with the combined size of the priority and the size of each queen's state.
    3. Calculate the memory size (explored_size) by summing up the size of each state in the explored set.
    4. Return a tuple of (frontier_size, explored_size).

13. Astar.run(problem)

- Input: *problem* - An instance of the NQueensProblem class.
- Output: Tuple containing the solution node, frontier size, and explored size.
- Description: Executes the A* algorithm to find a solution for the N-Queens problem. It expands nodes based on their combined path cost and heuristic cost and maintains a priority queue (frontier) for efficient exploration.

- Procedure:
  1. Initialize the frontier and explored set.
  2. Create a node with the initial state, path cost of 0, and heuristic cost based on the initial state.
  3. Push the node onto the frontier using the combined cost of path cost and heuristic cost as the priority.
  4. Enter a loop until a goal state is found or the frontier becomes empty.
  5. If the frontier is empty, return None to indicate that no solution was found.
  6. Pop the node with the lowest combined cost from the frontier.
  7. Print the node's state to visualize the search process.
  8. Check if the node's heuristic cost is 0. If so, return the node and memory usage.
  9. Add the node's state to the explored set.
  10. Generate child nodes by applying actions to the current node's state.
  11. For each child node:
      a. If its state is not in the explored set and not in the frontier, push it onto the frontier.
      b. Else if it is in the frontier but has a lower combined cost, update the corresponding node in the frontier.

14. Astar.childNode(parent,action,problem)

- Input:
  - *parent* - The parent node from which the child node is derived.
  - *action* - The action representing the movement of a queen to a new row in the same column.
  - *problem* - An instance of the NQueensProblem class representing the N-Queens problem.
- Output: A new instance of the NodeAstar class representing the child node.
- Description: This function generates a child node from a parent node by copying the parent's state and updating the row of a queen based on the action. The path cost of the child node is updated by incrementing the parent's path cost by 1. and the heuristic cost of the child node is calculated using the problem.heuristic() method.

- Procedure:
    1. Create a copy of the parent node's state to modify.
    2. Apply the action to the child state by updating the queen's position in the corresponding column.
    3. Calculate the heuristic cost for the child node based on the changed state by invoking the problem's heuristic_change method.
    4. Create a new NodeAstar instance for the child node with the updated state, a path cost incremented by 1, and the calculated heuristic cost.
    5. Return the child node.

15. Astar.get_memory_usage()

- Input: None
- Output: Tuple containing the frontier size and explored size in megabytes (MB).
- Description: Calculates the memory usage of the A* algorithm. The function estimates the memory occupied by the frontier (priority queue) and the explored set.
- Procedure:
    1. Get the length of the frontier list and the number of queens (n_queens).
    2. Because the frontier store item in structure: (cost, node) so the memory size (frontier_size) can be calculated by multiplying the length of the frontier with the combined size of the priority and the size of each queen's state.
    3. Calculate the memory size (explored_size) by summing up the size of each item in the explored set.
    4. Return a tuple of (frontier_size, explored_size).

16. GeneticSearch.run(problem)

- Input: *problem* - An instance of the NQueensProblem class.
- Output: Tuple containing the solution node and population memory.
- Description: Executes the Genetic Search algorithm to find a solution for the N-Queens problem. It evolves a population of nodes through reproduction, mutation, and selection processes.
- Procedure:
    2. Initialize the population as an empty list.

3. Generate an initial population of nodes with a population size equal to the number of queens * 2, each with random states and corresponding heuristic costs.
4. Enter a loop until a goal state is found or the number of generations exceeds 100000
5. Check each node in the population for the goal condition. If a goal state is found, calculate the memory usage and return the node along with the population memory.
6. Select pairs of parents from the population based on their fitness.
7. Create a new population by reproducing the selected parents and mutating them at a specific rate.
8. Update the population with the new generation of nodes.
9. Print the generation number and the minimum heuristic cost of the population to visualize the search process
10. Increment the generation counter and continue the loop.

17. GeneticSearch.select_parents(population)

- Input: *population* - The current population of nodes.
- Output: A list of tuples, where each tuple contains two parent nodes.
- Description: The selection process uses a roulette wheel approach, where parents are chosen with probabilities proportional to their fitness scores. The selected parents are returned as a list of tuples, with each tuple containing two parent nodes for reproduction.
- Procedure:
  1. Sort the population based on the heuristic cost of each node.
  2. Calculate the sum of fitness values (1/heuristic_cost) for all nodes in the sorted population.
  3. Iterate over the population size to choose two parents based on their fitness scores for each iteration.
     a. Generate two random numbers (random1, random2) between 0 and 1.
     b. Initialize variables parent1 and parent2 as None.
     c. Calculate the cumulative fitness by iterating over the sorted population and adding the fitness value.

    d. Check if random1 is less than or equal to cumulative_fitness / sum_fitness for each node in the sorted population. If true, set parent1 as the current node and break the loop.
    e. Repeat the same process for random2 to determine parent2.
    f. Append a tuple of parent1 and parent2 to the parents list.
  4. Return the list of selected parent pairs.

18. **GeneticSearch.reproduce(parent1, parent2, problem)**

- Input:
    - *parent1* - The first parent node.
    - *parent2* - The second parent node.
    - *problem* - An instance of the NQueensProblem class.
- Output: A new instance of the NodeGenetic class representing the child node.
- Description: Reproduces a child node from two-parent nodes through a crossover operation. The function randomly selects a crossover point, combines the genetic material (state) of the parents before and after the crossover point, and creates a new state for the child node. The child node's heuristic cost is evaluated using the problem.heuristic() method.
- Procedure:
    1. Generate a random index between 0 and the length of the parent1's state minus 1.
    2. Create a child state by concatenating the prefix of parent1's state up to the random index with the suffix of parent2's state starting from the random index.
    3. Create a new NodeGenetic instance for the child node with the created state and the heuristic cost calculated based on the child state.
    4. Return the child node.

19. GeneticSearch.mutate(node,problem)

- Input:
    - *node* - The node to be mutated.
    - *problem* - An instance of the NQueensProblem class.
- Output: A new instance of the NodeGenetic class representing the mutated node.

- Description: Mutates a node by randomly changing the row position of a randomly selected queen. The function selects a random index representing the queen, generates a new random position, and updates the state accordingly. The function recalculates the node's heuristic cost using the problem.heuristic_change() method.
- Procedure:
    1. Generate a random index between 0 and the length of the node's state minus 1.
    2. Generate a new random position for the queen in the state.
    3. Ensure that the new position is different from the original position by checking that it is not equal to node.state[index].
    4. Update the node's state by replacing the queen's position at the random index with the new position.
    5. Update the heuristic cost of the node by invoking the problem's heuristic_change method.
    6. Return the updated node.

20. GeneticSearch.get_memory_usage()

- Input: None
- Output: Population memory in megabytes (MB).
- Description: Calculates the memory usage of the Genetic Search algorithm. The function estimates the memory occupied by the population of nodes.
- Procedure:
    1. Get the size of the population (population_size) and the number of queens (n_queens).
    2. Calculate the memory size (population_memory) by multiplying the population size with the combined size of each queen's state and the heuristic cost.
    3. Return the population_memory value.

4. Performance of Search Algorithms

4.1. Computer Configuration

- Processor: Intel Core i5-11400H @ 2.70 GHz (12CPUs)
- RAM: 16GB

4.2. Runtime and Memory Measurement

To evaluate the performance of the program, both runtime and memory usage are measured during its execution.

4.2.1. Runtime

The runtime of the program is measured using the %%time magic command, which is executed before the main program execution. This command records the elapsed time from the start to the completion of the program. The recorded runtime includes the time taken by all the executed statements, including user inputs, algorithm execution, and output display.

4.2.2. Memory

The program’s memory usage is measured using the sys module, specifically the getsizeof() function. Memory usage is calculated by tracking the size of relevant objects, such as data structures and variables, during the program's execution. In the case of search algorithms, memory usage is primarily influenced by the size of the frontier (priority queue), explored set, and other relevant data structures. The sys.getsizeof() function is applied to estimate the memory usage of these objects.

- For UCS and A* algorithms, the memory usage is calculated by measuring the size of the frontier and explored set. The size of each element in the frontier or explored set is determined using sys.getsizeof(), and the total memory is computed by summing up the sizes of all elements.
- For the Genetic Search algorithm, memory usage is measured by estimating the memory occupied by the population of nodes. The size of each node, including the state and heuristic cost, is determined using sys.getsizeof(). The total memory is calculated by multiplying the number of nodes in the population by the size of each node and summing up these values.

4.3. Measurement Results

	Running time (ms)			Memory (MB)		
Algorithms	N = 8	N = 100	N = 500	N= 8	N = 100	N= 500
UCS	Intractable	Intractable	Intractable	Intractable	Intractable	Intractable
A*	1042ms	Intractable	Intractable	0.415	Intractable	Intractable
GA	101.45ms	11min 27000ms	Intractable	0.0038	0.5394	Intractable

4.4. Explanation

4.4.1. Uniform Cost Search

Uniform-cost search (UCS) is an algorithm used to explore the search space by considering the cost of reaching each state. In the case of the N-queens problem, UCS aims to find a sequence of moves or placements for the queens that minimize the total cost of moving the queens from their initial positions to a valid solution.

In UCS, the cost of moving a queen is defined as 1. Therefore, the time and space complexity of UCS is $O(b^{1+d})$, where b is the branching factor and d is the depth of the search space. In this specific case, the branching factor can be defined as b = n(n-1). As the number of queens (N) increases, the search space grows exponentially. With more queens, there are a significantly larger number of possible configurations to explore, resulting in a vast number of states to evaluate. The algorithm needs to traverse and store information about all visited nodes and edges during the search process. This exponential growth in the search space makes it increasingly challenging, time and memory-consuming for UCS to find a solution.

In the experiments conducted, when running UCS with n=6, it took approximately 1min 52s and consumed 4.37 megabytes of memory. However, when n=7 or higher, UCS becomes intractable due to the extensive time it takes to find a solution (more than 12 hours) Furthermore, as n continued to grow even larger, an additional challenge emerged related to memory limitations. The UCS algorithm could potentially encounter memory overflow issues when n reaches higher values, risking exceeding the available RAM capacity.

A small improvement that can be applied to UCS, in this case, is to omit the step of updating the path cost for nodes already in the frontier with a lower path cost. In the N-Queens problem, where the cost of each path is the same (equal to 1), the UCS algorithm approximates the BFS (Breadth-first Search) algorithm. Therefore, the path cost update step will never be executed, allowing us to omit it and reduce some time costs.

Another adjustment is to not check whether a node is in the frontier or not, and simply add it to the frontier. Since the frontier uses a heapq module with the priority being the path cost, it will automatically select the node with the lowest cost to add to the explored set. Later, if a node is retrieved and found to be in the explored set, it can be skipped and the algorithm can continue. This approach can help save a significant amount of runtime by avoiding the cost of searching within the frontier (with n = 8, the average runtime is 25.3s). However, it comes with the trade-off of increased memory usage (over 600 MB with n = 8) because the frontier will have a larger size, potentially posing a higher risk of memory overflow compared to the original version of the algorithm as n increases.

4.4.2. A* Search

A* search is an informed search algorithm that uses a heuristic function to guide its exploration. In the context of the N-queens problem, A* with the MIN-CONFLICT heuristic aims to find a sequence of moves for the queens that minimize the number of attacking pairs of queens. The heuristic function provides an estimate of how far a state is from the goal state based on the number of attacking pairs. By using the MIN-CONFLICT heuristic, A* prioritizes exploring states with fewer attacking pairs, as these states are closer to a valid solution.

The time and space complexity of the A* algorithm is known to be exponential. As the number of queens (N) increases, the complexity of the N-queens problem grows exponentially. The number of possible configurations to evaluate becomes extremely large, leading to an exponential increase in the number of attacking pairs. This exponential growth in the search space makes it increasingly challenging for A* to find an optimal solution. While the MIN-CONFLICT heuristic guides reducing the number of attacking pairs, the exponential increase in the number of possible configurations makes it difficult for A* search to explore all paths and find an optimal solution within a reasonable time frame.

In the experiments conducted, when running A* with n=20, it took approximately 15min37s and consumed 17.2575 megabytes of memory. However, when n>20 or higher, A* becomes intractable due to the extensive time it takes to find a solution (more than 12 hours). Although the A* algorithm does not consume as much memory as UCS, when the number of queens increases significantly, memory-related issues can arise, posing a risk of memory overflow.

The A* algorithm can also be adjusted similarly to the UCS algorithm by omitting the step of updating the path cost for nodes already in the frontier. This can be done because the path cost is fixed to 1 and the heuristic function is constant for each node (as the number of attacking queens remains the same for each node) in the N-Queens problem. Therefore, the update step will never be executed. This adjustment helps to reduce the running time slightly, as mentioned above.

The second adjustment can also be applied to the A* algorithm. We no longer need to check whether a node is in the frontier or not before adding it. Instead, we simply add the node to the frontier. Since the frontier is implemented using a heapq module, where the priority is determined by the path cost, it will automatically select the node with the lowest cost to be added to the explored set. Later, when a node is retrieved and found to be in the explored set, it can be skipped, and the algorithm can proceed. This approach can significantly reduce the runtime by avoiding the cost of searching within the frontier (with n = 20, the average runtime is 1.4185s). However, it comes at the expense of increased memory usage (over 200 MB with n = 8), as the frontier will have a larger size. This increased memory usage can potentially pose a higher risk of memory overflow compared to the original version of the algorithm. Especially when n = 100, although the algorithm runs within the allowed time, in most states it becomes overloaded and experiences a memory overflow before finding a solution.

4.4.3. Genetic Algorithm

Genetic algorithms are a class of optimization algorithms inspired by the process of natural selection. In the context of the N-queens problem, a genetic algorithm is used to find a solution that minimizes the number of attacking pairs of queens. The genetic algorithm starts with a population of randomly generated configurations, where each configuration represents a possible arrangement of the queens on the chessboard. The algorithm then applies a series of genetic operators, including selection, crossover, and mutation, to evolve the population over multiple generations.

During the selection process, configurations with a lower number of attacking pairs are more likely to be chosen as parents for the next generation. This bias towards better solutions allows the algorithm to gradually improve the population over time.

Crossover is performed by taking two parent configurations and creating offspring configurations by combining their genetic material (queen positions). This helps explore new configurations and potentially discover better solutions.

Mutation introduces random changes to the configurations, allowing the algorithm to explore different parts of the search space. It helps prevent the algorithm from getting stuck in local optima and promotes diversification within the population.

The genetic algorithm evaluates the fitness of each configuration in the population based on the number of attacking pairs. Configurations with a lower number of attacking pairs have higher fitness and are more likely to be selected for the next generation.

As the number of queens (N) increases beyond 100 the Genetic search becomes intractable due to the extensive time it takes to find a solution (more than 12 hours). Because the search space becomes exponentially larger, exploring all possible configurations becomes more computationally expensive. The computational resources required to evaluate a large number of configurations and perform the genetic operators can become a limiting factor. The computational complexity is also influenced by the population size and mutation rate. A larger population size helps diversify the problem states and avoid local optima, but it comes with limitations such as increased computational cost, runtime, and memory usage. A higher mutation rate helps the algorithm explore unexplored regions in the search space, avoiding local optima and maintaining diversity in the population, but it can also lead to rapid loss of information (losing discovered good features), generating poor solutions, reducing convergence to optimal solutions, and increasing runtime due to recalculating the number of queens attacking each other with each mutation. Therefore, the selection of the population size and mutation rate must be appropriate to strike a balance and optimize the algorithm. In the genetic search algorithm used, the population size is set to be twice the number of queens, and the mutation rate is typically between 60% and 80%.

To ensure computational efficiency and prevent excessive runtime, the genetic algorithm stops at generation 100,000. This balanced limit allows for a substantial number of iterations while avoiding indefinite execution. It safeguards against situations where the algorithm would run endlessly without a solution, particularly in cases with only 2 or 3 queens.

5. References

1. Slide from Lecture 03, Part 2, on Uninformed Search:

Nguyen, Ngoc Thao, and Nguyen, Hai Minh. "2021-Lecture03-P2-UninformedSearch." Artificial Intelligence Course. Lecture slide.

2. Slide from Lecture 03, Part 3, on Informed Search:

Nguyen, Ngoc Thao, and Nguyen, Hai Minh. "2021-Lecture03-P3-InformedSearch." Artificial Intelligence Course. Lecture slide.

3. Slide from Lecture 04 on Local Search:

Nguyen, Ngoc Thao, and Nguyen, Hai Minh. "2021-Lecture04-LocalSearch." Artificial Intelligence Course. Lecture slide.