TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

21127166 – NGUYỄN TUẤN THANH

LAB01: N-QUEENS PROBLEM

Tp. Hồ Chí Minh, tháng 07/2023

Contents:

I. N – QUEENS PROBLEM

1. Description

- You are considering the N - queens problem, in which:
  - The complete state formulation represents a configuration, i.e., all the N queens are on the board. Furthermore, each queen is on a separate column.
  - The successors of any state are all possible configurations generated by moving a single queen to another square in the same column.

2. Formulation

- State Space: The state space represents all possible configurations of the chessboard with N queens. The size of the state space is at least $(N!)^{1/3}$, which accounts for the lower bound. However, it may be much larger depending on the specific problem instance and the constraints applied.
- Initial State: The initial state is a board with N queens placed on different columns in such a way that no two queens attack each other. The specific arrangement of the queens can be randomly generated or based on some predefined rules.
- Actions: The available actions correspond to moving a single queen to the leftmost empty column that does not result in any attacks between queens. Each action involves choosing a queen and moving it to a different row in its own column.
- Path Cost: The path cost associated with each action is 1 since we only care about reaching the final board configuration where all N queens are placed without attacking each other. The goal is to find a valid configuration, and the path cost does not play a role in the search algorithms.
- Transition Model: The transition model defines how a state is transformed by applying an action. In this case, the transition model involves moving a single queen to a different row within its own column. This operation generates a successor state by modifying the position of one queen while keeping the other queens fixed.
- Goal State: The goal state is reached when the chessboard is filled with N queens, with each queen placed on a different column and no two queens attacking each other. It represents a valid and optimal solution to the problem.

II. IDEAS

1. Data Structure

- We will use a list of integers to represent the state of the chessboard. Each element of the list corresponds to a column, and its value represents the row where a queen is placed. This representation allows us to easily check if two queens attack each other, and it aligns well with the genetic search algorithm, which also utilizes lists for crossover and mutation operations.

2. Code Clarification

- The given code defines two subclasses (AStarState and GeneticAlgorithmState) and a parent class (State) in Python. These classes are used to model the state of a specific problem and perform some related methods for handling that state.

a. Class State

- The __init__ (self, initial_state) method is the constructor of the class, which takes an initial_state parameter and assigns its value to the current_state attribute of the object.
- The count_conflicts (self, current_state) method calculates the number of conflicts in the current state. This is a method specific to the State class to compute the number of conflicts between pairs of queens in the NQueens problem. It iterates through each pair of queens and checks if they are on the same row, column, or diagonal. If there is a conflict, the conflict count is increased.
- The get_successors (self, current_state) method returns a list of the next possible states from the current state. It iterates through each queen in the state and replaces each queen with different possible positions. Then, it adds these new states to the successors list and returns the list.
- The goal_test (self, current_state) method checks whether the current state is a goal state or not. It checks if there are any two queens on the same row, column, or diagonal. If there is a conflict, the state is not a goal state and returns False. If there are no conflicts, the state is a goal state and returns True.
- Class AStarState (inherits from State):
    - This class extends the State class and adds a heuristic attribute. The __init__ (self, initial_state) constructor of this subclass calls the constructor of the parent class and then assigns the value of 0 to the heuristic attribute.
- Class GeneticAlgorithmState (inherits from State):

- This class also extends the State class but does not add any new attributes. The __init__ (self, initial_state) constructor of this subclass calls the constructor of the parent class and does not perform any additional actions.

b. Class Search

- This class serves as the parent class for different search algorithms. It has an __init__ (self, initial_state: State) method that initializes the state and explored set.
- The solve (self) method is declared but not implemented. Subclasses of Search are expected to implement this method.
- **Class UniformCostSearch** (inherits from Search)
  - This class represents the Uniform Cost Search algorithm. It extends the Search class and overrides the __init__ (self, initial_state) and solve(self) methods.
  - The __init__ (self, initial_state) constructor calls the constructor of the parent class and initializes the frontier using a priority queue (heapq). The frontier stores tuples of the form (cost, current_state) where cost represents the cost to reach current_state.
  - **The solve () method** starts by initializing the initial state and creating a frontier as a priority queue with the initial state and a path cost of 0. The algorithm continues if the frontier is not empty. At each iteration, it selects the lowest-cost node from the frontier. If the selected node is a goal state, it returns to the current state, frontier, and explored states. Otherwise, it adds the current state to the explored set and expands the current state by generating successors. For each successor, it checks if the successor is not in the explored set. If the successor is already in the frontier, it compares the existing node's path cost with the current path cost. If the current path cost is lower, it replaces the existing node with the successor. Finally, it adds the successor to the frontier with an updated cost and heapifies the frontier. The algorithm prints the size of the frontier at each iteration. If no goal state is found, it returns None.
- **Class AStarSearch** (inherits from Search):
  - This class represents the A* Search algorithm. It extends the Search class and overrides the __init__ (self, initial_state: AStarState) and solve(self) methods.
  - The __init__ (self, initial_state: AStarState) constructor calls the constructor of the parent class and initializes the frontier using a priority queue (heapq). The frontier stores tuples of the form (heuristic, cost, current_state) where heuristic represents the heuristic value of current_state.
  - **The solve () method** initializes the initial state and creates a frontier as a priority queue with a tuple representing the current state's count of conflicts, path cost, and

current state. The algorithm continues if the frontier is not empty. It selects the node with the lowest heuristic value from the frontier. If the selected node has a heuristic value of 0, it returns the current state, frontier, and explored states. Otherwise, it adds the current state to the explored set, expands the current state by generating successors, and performs the same checks and updates as in the Uniform Cost Search algorithm. Additionally, it prints the heuristic value of the first node in the frontier at each iteration. If no goal state is found, it returns None.

- **Class GeneticAlgorithm** (inherits from Search):
  - The __init__ (self, initial_state: GeneticAlgorithmState) constructor initializes the GeneticAlgorithm class with the initial state and a population containing only one individual the initial state and sorts the population based on the fitness value.
  - The initialize_population (self, num_queens) method initializes the population for the Genetic Algorithm. It generates a random number of individuals between 2 and the number of queens. For each individual, it creates a list of random positions for the queens. The fitness of each individual is evaluated using the fitness method. The population is stored as a heap using the heapq module.
  - The fitness (self, current_state) method calculates the fitness value of a state based on the number of collisions between the queens. A lower fitness value indicates a better state.
  - The random_pick(self) method randomly selects a subset of individuals from the current population. The number of individuals selected randomly falls within the range of 2 to the size of the population.
  - The crossover (self, parent1, parent2) method performs the crossover process between two parent individuals and randomly selects a crossover point and creates two offspring individuals by exchanging genes from the crossover point.
  - The mutate (self, child) performs the mutation process on an individual and randomly selects a gene and changes its value to a random value.
  - **The solve () method** performs the Genetic Algorithm search algorithm. Initially, it initializes the population and checks if the initial individual is the goal state. Next, in each generation, it randomly selects a subset of individuals from the current population, performs crossover and mutation processes, updates the population, and checks if there is a goal state. This process continues until a goal state is found or there are no individuals left in the population.

III. PERFORMANCE OF ALGORITHMS

1. Measuring running times and consumed memory

	Running time (ms)			Memory (MB)		
Algorithm	**N = 8**	**N = 100**	**N = 500**	**N = 8**	**N = 100**	**N = 500**
UCS	Intractable	Intractable	Intractable	Intractable	Intractable	Intractable
A*	29.4197	Intractable	Intractable	0.0823	Intractable	Intractable
GA	196.6263	1162533.199	Intractable	0.0006	0.0110	Intractable

2. Explanation

2.1 Uniform cost search

a. Drawback

- Large State Space: With the n-queens problem, the state space can be extremely large. The number of feasible states can reach up to $(N!)^{3/2}$ where each queen can have up to n different positions on each row and column. Therefore, exploring the entire state space to search for a solution can be very time-consuming.
- Uniform Cost Expansion: Uniform Cost Search expands states in a uniform cost order, incrementally increasing the cost at each step. This means that expanding all possible states can take a significant amount of time and computational resources. The incremental steps in expanding states can slow down the algorithm, especially when reaching the later stages of the solution.
- Limited Cost Evaluation: Uniform Cost Search does not use a heuristic function to estimate the cost from the current state to the goal state. Instead, it relies solely on the local cost of each movement step. In the n-queens problem, the lack of estimated information can lead to inefficient search paths and blind exploration.

b. Optimization approach

- For the n-queens problem, the calculation of PATH_COST is not necessary. In the Uniform Cost Search algorithm, the PATH_COST increases uniformly by one unit at each step, and the algorithm always selects the state with the smallest cost from the frontier. Therefore, there is no need to explicitly update the PATH_COST during the search. By omitting the calculation and updating of PATH_COST, the algorithm reduces the computational overhead and avoids unnecessary operations. This

optimization allows the algorithm to concentrate its resources on exploring the state space and finding valid solutions efficiently.

2.2 A* search

a. Drawback

- Time and Space Complexity: A* Search can be time-consuming and require large memory space for large n-queens problems. This is because the number of feasible states in the search space increases exponentially with the size of the problem. The space complexity of A* Search is still lower than that of Uniform Cost Search because A* utilizes a heuristic function to efficiently approach the goal state.
- Handling Duplicate States: During the search process, A* Search may encounter duplicate states. Checking and removing duplicate states can take time and memory, affecting the algorithm's performance.
- Exploring the Entire State Space: A* Search requires traversing the entire state space to find the optimal solution. This becomes particularly challenging for large n-queens problems, where the number of feasible states increases significantly, resulting in long computation time and impracticality in real-world scenarios.

b. Optimization approach

- For the n-queens problem, the calculation of PATH_COST is not necessary. In the Uniform Cost Search algorithm, the PATH_COST increases uniformly by one unit at each step, and the algorithm always selects the state with the smallest cost from the frontier. Therefore, there is no need to explicitly update the PATH_COST during the search. By omitting the calculation and updating of PATH_COST, the algorithm reduces the computational overhead and avoids unnecessary operations. This optimization allows the algorithm to concentrate its resources on exploring the state space and finding valid solutions efficiently.

2.3 Genetic Algorithm

a. Drawback

- Dependency on Parameters: Genetic Algorithm relies heavily on parameter settings, such as population size, mutation rate, crossover rate, and the number of generations. These parameters play a crucial role in shaping the behavior and effectiveness of the algorithm. Choosing appropriate values for these parameters is essential to achieve good performance and optimal results. However,

determining the optimal parameter values can be challenging and may require experimentation and domain knowledge.

- Prone to Local Optima: Genetic Algorithm is susceptible to getting trapped in local optima, which are suboptimal solutions within the search space. If the algorithm converges to a local optimum, it may fail to explore other regions of the search space that could potentially contain better solutions. The risk of falling into local optima increases when the population lacks diversity or when the genetic operators, such as crossover and mutation, are not sufficiently explorative or exploitative.
- No Guarantee of Global Optimal Solution: Genetic Algorithm does not provide a guarantee of finding the global optimal solution for a given problem. The algorithm explores the search space through evolutionary processes, but it does not guarantee that the best possible solution will be discovered. The outcome of the algorithm depends on the initial population, parameter settings, and the characteristics of the problem being solved. Multiple runs with different configurations may be needed to improve the likelihood of finding a good solution.

b. Optimization approach

- If the initial population size is 2n, the Genetic Search algorithm can work more effectively because this population size allows for greater diversity and richness, preventing the problem from getting trapped in local optima.
- Instead of using the fitness function individually for each element in the population and then running a random selection function, a better approach is to sort the population based on the heuristic value from small to large and use functions such as logistic, exponential, or parabolic functions. With this approach, the time complexity of the selection operation is reduced from $O(n^2)$ (using the fitness function) to $O(n \log_2 n)$ for each element selected in a generation.
- These improvements can help enhance the exploration and exploitation capabilities of the Genetic Search algorithm, increasing the chances of finding better solutions and improving its efficiency.

IV. REFERENCES

[1]: https://compsci.sites.tjhsst.edu/ai/nqueens/NQueens.pdf

[2]: https://youtu.be/QXn27pUyuNQ

[3]: http://www.sc.ehu.es/ccwbayes/docencia/kzmm/files/AG-nQueens.pdf

[4]: https://youtu.be/Fgq3xDhRBv8

[5]: Artificial Intelligence A Modern Approach - 3rd Edition