N-Queens Searching Strategies

Name: Châu Tấn Kiệt
Class: 21CLC04
ID: 21127329

1/ Checklist

Requirements	Status
Complete-state formulation	Completed
Uniform-cost search	Completed
A*	Completed
Genetic Algorithm	Completed
8 Queens	Completed
100 Queens	Incomplete
500 Queens	Incomplete
GUI	Completed
Report	Completed

2/ Performance Table:

	Running time (ms)			Memory (MB)		
Algorithms	N = 8	N=100	N=500	N = 8	N=100	N=500
UCS	97526.31	Intractable	Intractable	420.0253	Intractable	Intractable
A*	16.80	Intractable	Intractable	0.518	Intractable	Intractable
Genetic	9102.34	Intractable	Intractable	0.0404	Intractable	Intractable

It can be seen that UCS is significantly slower than A* because it doesn't take the heuristic value of each state into account, but instead it only counts path cost, which in this case, is 1, making it equivalent to Breadth-First Search (BFS)

The A* search strategy has the shortest runtime due to using a good admissible heuristic function (h(x) <= h*(x))

3/ User Interface

```
1. A* Search
2. UCS Search
3. Genetic Algorithm
Choose an algorithm to solve N-queens: 1
Testcase no.1 :
* * * Q * * * *
* Q * * * * * *
* * * * Q * * *
* * * * * * * Q
* * * * * Q * *
Q * * * * * * *
* * Q * * * * *
* * * * * * Q *
Runtime of testcase no.1 : 27.86 ms
Memory of testcase no.1 : 0.1760 MB
Testcase no.2 :
* * * * Q * * *
* Q * * * * * *
* * * Q * * * *
* * * * * * Q *
* * Q * * * * *
* * * * * * * Q
* * * * * Q * *
Q * * * * * * *
Runtime of testcase no.2 : 14.14 ms
Memory of testcase no.2 : 0.0835 MB
Testcase no.3 :
* * Q * * * * *
* * * * * Q * *
* Q * * * * * *
* * * * * * Q *
Q * * * * * * *
* * * Q * * * *
* * * * * * * Q
* * * * Q * * *
Runtime of testcase no.3 : 8.41 ms
Memory of testcase no.3 : 0.0383 MB

Average time: 16.80 ms
Average memory usage: 0.0992 MB.
```

4/ Main explanations for code

- **A* and UCS:**

```python
def actions(self, state) -> bool:
        action = []
        for row in range(self.n):
            for col in range(self.n):
                if col != state[row]:
                    new_action = list(state[:])
                    new_action[row] = col
                    action.append(tuple(new_action))
        return action

    def result(self, state, action):
        newState = list(state[:])
        newState[action[0]] = action[1] # Location of a queen on a column
        return tuple(newState)

    def conflict_check(self, r1, c1, r2, c2): #Check vertically, horizontally and diagonally
        return r1 == r2 or c1 == c2 or abs(r1 - r2) == abs(c1 - c2)

    def goal_test(self, state: 'tuple[int]') -> bool:
        if self.h(state):
            return False
        return True

    def g(self, from_state, to_state):
        return self.h(to_state)

    def h(self, state: 'tuple[int]'):
        conflict_count = 0
        for i in range(len(state) - 1):
            for j in range(i + 1, len(state)):
                if self.conflict_check(i, state[i], j, state[j]): # If conflict occurs
                    conflict_count += 1                           # add 1 to the current conflict value
        return conflict_count
```

- conflict_check: check if there are queens attacking each other (diagonally, horizontally, and vertically)
- goal_test: this function check if the current state of the board is final (No 2 queens attacking each other)
- actions: returns every possible action in the current state.
- result: returns the next state that can be achieved from performing an action (Moving a queen)

- g: return the path cost from one state to another (can be 1)

- h: calculate the heuristic value of the current state, which is the min-conflict heuristic: "the number of attacking pairs of queens".

- **Genetic Algorithm:**

```python
def __init__(self, problem: Problem) -> None:
   self.problem = problem
   self.n = problem.n
   self.max_fitness = (self.n * (self.n - 1)) // 2
   self.population = [problem.random_state() for _ in range(Genetic.POPULATION_SIZE)]
   self.gen = 0
   self.solution = None

 def fitness(self, chromosome):
   return self.max_fitness - self.problem.h(chromosome)

 def probability(self, fitness): #Calculate probabilty
   return fitness / self.max_fitness
```

- With n queens on a n x n chessboard, there will be a maximum of n(n-1)/ 2 pairs of queens attacking each other, or we can say n (n/1) / 2 conflicts, so we can calculate the fitness value of each state.