Introduction to AI
NQueens with UCS, A* and Genetic Algorithm

Lab 1

21CLC05 - University of Science - VNUHCM

21127135 – Diệp Hữu Phúc

✦

1. Report

This will be a long read so please make use of bookmarks for ease of navigation.

Priority	Task	Status
Required	UCS Complete-state formulation	Done
	A* Complete-state formulation	Done
	Genetic Algorithm	Done
	Performance study	Done
	IPython Notebook	Done
Bonus	UCS Incremental formulation	Done
	A* Incremental formulation	Done

2. Implementation

The board is represented as an array of integers where each index serves as a column and the value of that index a row. This is also the portrayal of a state.

NQueens.py – Problem is expressed as classes for an OOP approach.

searchAlgo.py – All the drivers for the searches are written here.

UCS and A*

- During execution, the program will first ask whether the queens should be placed incrementally or not.
- There are 2 versions of the handleSearch function, one has **expanded** as a **list** and the other a **set**.

- The **list** version runs slower but saves more memory while the **set** one is the exact opposite.
- UCS should always use the latter, otherwise it will literally take ***an eternity*** to find a solution, especially in Complete-state formulation.

The only real difference, at least in my implementation, between UCS and A* is in the function that decides the ordering of nodes in the priority queue.

UCS	`def __lt__(self, node):` `  return self.pathCost < node.pathCost`
A*	`def __lt__(self, node):` `  return (self.pathCost + self.heuristic) < (node.pathCost + node.heuristic)`

For that reason, I will only elaborate on A* since it is more comprehensive.

Set up

```
def calcHeuristic(self, state) -> int:
    conflicts = 0
    for c1, r1 in enumerate(state):
        for c2, r2 in enumerate(state[c1 + 1:], c1 + 1):
            if self.checkConfict(r1, c1, r2, c2): conflicts += 2
    return conflicts
```

The heuristic of a state is the number of conflicts in the board. If queen A collides with queen B then the reverse is also true, that's why it will be counted as 2 conflicts. With the requirement being min-conflict heuristic, the solution – goal state – would have a heuristic of 0.

```
def AstartSearch(problem : NQueens) -> Node:
    node = NodeAstar(problem.initState,
heuristic=problem.calcHeuristic(problem.initState))
    return handleSearch(node, problem)

def handleSearch(node: Node, problem: NQueens) -> Node:
    frontier = [node]
    heapq.heapify(frontier)
    expanded = [problem.initState]
    while frontier:
        current = heapq.heappop(frontier)
        if problem.goalTest(current.state): return current
        if current in expanded: continue
        children = current.expand(problem)
        expanded.append(current)
        for i in children:
            if i not in expanded: heapq.heappush(frontier, i)
    return Node((), None, None)
```

handleSearch in *searchAlgo.py* is a generic (almost template) graph-search function. However, it is the main driver calling other components such as node.expand(problem) to push the search forward.

```
def uniformCostSearch(problem : NQueens) -> Node:
    node = NodeUCS(problem.initState)
    return handleSearchSet(node, problem)

def handleSearchSet(node: Node, problem: NQueens) -> Node:
    expanded = set()
    frontier = [node]
    heapq.heapify(frontier)
    expanded.add(problem.initState)
    while frontier:
        current = heapq.heappop(frontier)
        if problem.goalTest(current.state): return current
        children = current.expand(problem)
        for i in children:
            if i.state not in expanded:
                heapq.heappush(frontier, i)
                expanded.add(i.state)
    return Node((), None, None)
```

handleSearchSet is a variation of the original function where expanded is a set instead of a list. The reason for this is during **Performance study**, UCS would refuse to arrive at any solution even after 10 hours (I did this 6 times by running 6 instances of Google Colab simultaneously). This version of the function runs much faster but consumes significantly more memory.

```
def expand(self, problem) -> list:
    return [self.getChildNode(problem, action) for action in
problem.getActions(self.state)]
```

Expanding a node means we consider all the actions each queen could take – the rows that are possible for said queen to move to. With every single action, we can then generate a child node by placing the queen on that row.

```
def getChildNode(self, problem, action):
    nextState = problem.placeQueen(self.state, action)
    nextNode = NodeAstar(nextState, self, action,
problem.calcPathCost(self.pathCost), problem.calcHeuristic(nextState))
    return nextNode
```

Complete-state formulation

- Initially, all queens are placed on random rows. Each column is occupied by one queen.
- The solution varies depending on the initial state.
- UCS takes ***very long*** to find a solution.

```
def getActions(self, state) -> list:
    validActions = []
    for currCol in range(self.numQueens):
        for currRow in range(self.numQueens):
            if currRow != state[currCol]: validActions.append((currRow, currCol))
    return validActions
```

In this formulation, getting a list of possible actions is very simple since we will count all the rows of each column that are vacant, e.g., not occupied by a queen, as valid. However, problem arises when we realize that multiple queens can be moved, meaning the rows will overlap. We solve this by pairing the rows with their column, which gives us the output of an array of pair of integers, or in Python terms, a list of tuples.

```
def placeQueen(self, state, action) -> tuple:
    newState = list(state)
    newState[action[1]] = action[0]
    return tuple(newState)
```

With that out of the way, all we need to do is place the queen (of the corresponding column) on one of the valid rows and we will have ourselves a new state.

Incremental formulation

- Initially, all queens are placed on row -1. Each column is occupied by one queen.
- The solution for a specific N will always be the same no matter how many times it's run.

```
def getActions(self, state) -> list:
    if self.isIncremental:
        if state[-1] != -1: return []
        validActions = list(range(self.numQueens))
        currCol = state.index(-1)
        for currRow in range(self.numQueens):
            for c, r in enumerate(state[:currCol]):
                if currRow in validActions and self.checkConfict(currRow,
currCol, r, c):
                    validActions.remove(currRow)
        return validActions
```

For this formulation, we only consider one queen at a time and since all queens start at row -1, we just need to get the leftmost column that is still empty and proceed from there. Utilizing this advantage, we can narrow down the possible actions even further from *all the rows of the current column that are vacant* to *all the rows of the current column that are vacant and doesn't collide with previously placed queens (the queens on the left side of the current column)*.

```
def placeQueen(self, state, action) -> tuple:
    if self.isIncremental:
        col = state.index(-1)
        newState = list(state[:])
```

```
        newState[col] = action
        return tuple(newState)
```

Then a new state will be gotten by placing the queen on one of the valid rows of the leftmost empty column.

Genetic Algorithm

- Each generation has 100 chromosomes, there is an option to print out all generations with their chromosomes.
- The fittest chromosome is one where the number of attacking pairs of queens is 0. Mutation probability is set to 0.8.
- The algorithm takes *very long* to find a solution.

```
def handleGeneticAlgo(problem : NQueensGenetic) -> list:
    if not problem.chromDisplay: print("The program is not actually stuck. Please
wait...")
    while not 0 in [problem.calcFitness(chrom) for chrom in problem.population]:
        if problem.chromDisplay: print(f'== Generation {problem.generation} ==')
        problem.population = problem.geneticQueen(problem.population)
        problem.generation += 1
    for chrom in problem.population:
        if problem.calcFitness(chrom) == 0: return chrom
```

Above is the driver for this algorithm, it checks if the fittest chromosome (goal state) exists in the current population and returns it, otherwise it will call geneticQueen on said population to generate a new one.

```
    def calcFitness(self, chromosome) -> int:
        return self.calcHeuristic(chromosome)
```

Since the objective function is defined from the same min-conflict heuristic, where the goal state should have a heuristic of 0, this translates to the fittest chromosome also having a fitness value of 0. For that reason, I took the liberty to reuse the calcHeuristic function.

```
def geneticQueen(self, population):
    mutationProb = 0.8
    newPopulation = []
    probs = [self.calcProbability(chrom) for chrom in population]
    for _ in range(len(population)):
        ch1, ch2 = random.choices(population, weights=probs, k=2)
        child = self.reproduce(ch1, ch2)
        if random.random() < mutationProb: child = self.mutate(child)
        if self.chromDisplay: self.print_chromosome(child)
        newPopulation.append(child)
        if self.calcFitness(child) == 0: break
    return newPopulation
```

Now this is a textbook version of the genetic function, I just went about implementing all the contributing components.

```
def calcProbability(self, chromosome) -> float:
    return self.calcFitness(chromosome) / self.maxConflicts
```

I chose to compute the weight of a chromosome (probability of it being chosen) by simply dividing its fitness (conflicts) to the max possible conflicts that a N board can have.

```
self.maxConflicts = numQueens * (numQueens - 1) # nC2 * 2
```

The formula for this is $C_N^2 \times 2$, we choose 2 queens, if they collide with each other then we count that as 2 conflicts. Expand that further and we'll arrive at the concise equation $N \times (N - 1)$.

```
def reproduce(self, x, y) -> list:
    xLen = len(x)
    crossoverPoint = random.randint(0, xLen - 1)
    return x[0:crossoverPoint] + y[crossoverPoint:xLen]

def mutate(self, x) -> list:
    xLen = len(x)
    randInd = random.randint(0, xLen - 1)
    randVal = random.randint(0, xLen - 1)
    x[randInd] = randVal
    return x
```

There is nothing much to say about reproduce (crossover) and mutate functions since they are very simple, at least in Python, and self-documented. On a side note, I would like to add a fun fact that I did come across the notion of double mutation (the source will be included in the **Reference** section), which – just as the name suggests – may mutate the chromosome another time. However, I *really don't think* it matters much since, in the end, it's all luck-based and in the worst case, you'd still have to wait a *possible eternity*. This is also the justification for why I arbitrarily picked 0.8 as the probability of mutation.

3. Performance study

All the tests are done while complying perfectly with all the requirements, which are:

- Complete-state formulation: This is applied by having every initial placement of queens be randomized.
- Min conflicts – The number of attacking pairs of queens: Although I count 2 queens colliding as 2 conflicts instead of 1, it won't make any difference. In A*, the goal state will have a heuristic of 0. As for Genetic Algorithm, its fittest chromosome will have a fitness value of 0.

Algorithm	**Running time (ms)**			**Memory (MB)**		
	N = 8	*N = 100*	*N = 500*	*N = 8*	*N = 100*	*N = 500*

UCS	96891.0000	Intractable	Intractable	687.1767	Intractable	Intractable
*A**	183.9049	13770134.67	Intractable	0.7843	1532.7805	Intractable
Genetic AI	2609006.402	Intractable	Intractable	0.4636	Intractable	Intractable

- **With UCS**, I had no choice but to use the handleSearchSet function since the algorithm refused to output any solution even after 10 hours (I did this 6 times by running 6 instances of Google Colab simultaneously). This resulted in the massive memory consumption. Just with 8 queens, the runtime and memory were far from ideal, thus, I decided against proceeding with 100 or more queens.
- **A* is**, by a long shot, the fastest algorithm and the likeliest to yield a result for a large number of queens out of the three. However, even as reliable as it is, testing with 100 queens still depleted a significant amount of time and resources and I am sure that having that number fivefold is not possible given my currently available options.
- **Genetic Algorithm** is an odd one since theoretically, it could solve any number of queens, you just need to have enough luck. Well, while doing this study, luck was definitely not on my side since even with just 8 queens, the time it drained are embarrassing. Despite that, I did still attempt 3 runs of 100, all of which ended with Google Colab failing me after 7 hours in.

UCS - handleSearchSet

N	Run	Platform	Initial State	Solution	Runtime (ms)	Memory (MB)
8	*1*	Console	[0, 7, 6, 3, 5, 2, 7, 5]	[2, 4, 6, 0, 3, 1, 7, 5]	253750.0000	1600.3246
	2	Console	[5, 2, 6, 5, 7, 6, 4, 7]	[5, 3, 6, 0, 7, 1, 4, 2]	8110.0000	109.4099
	3	Console	[1, 2, 4, 6, 5, 3, 3, 5]	[5, 2, 4, 6, 0, 3, 1, 7]	28813.0000	351.7957
	Average				96891.0000	687.1767
100	*0*	Intractable				
500	*0*	Intractable				

A*

N	Run	Platform	Initial State	Solution	Runtime (ms)	Memory (MB)
8	*1*	Colab	[3, 0, 6, 6, 5, 5, 4, 7]	[3, 1, 6, 2, 5, 7, 4, 0]	309.2167	1.3085
	2	Colab	[0, 3, 5, 2, 5, 5, 5, 3]	[7, 3, 0, 2, 5, 1, 6, 4]	217.4530	0.9889
	3	Colab	[1, 5, 3, 5, 0, 6, 5, 6]	[3, 1, 7, 5, 0, 2, 4, 6]	25.0451	0.0555
	Average				183.9049	0.7843

100	*1*	Colab	(A101i)	(A101s)	8204774.3169	902.2925
	2	Colab	(A102i)	(A102s)	11229832.1566	1229.1141
	3	Colab	(A103i)	(A103s)	21875797.5301	2466.9348
	Average				13770134.67	1532.7805
500	*0*	Intractable				

(A101i) [38, 71, 85, 91, 32, 84, 74, 64, 34, 0, 96, 25, 33, 61, 34, 5, 74, 70, 6, 15, 38, 54, 53, 24, 95, 89, 36, 95, 68, 31, 16, 56, 57, 36, 33, 4, 31, 33, 1, 61, 35, 53, 89, 44, 74, 97, 5, 17, 72, 15, 29, 91, 77, 26, 36, 31, 73, 81, 96, 4, 59, 44, 66, 36, 46, 48, 51, 50, 29, 72, 8, 55, 99, 72, 44, 81, 8, 78, 80, 67, 80, 92, 53, 20, 66, 3, 96, 69, 80, 85, 24, 68, 29, 72, 62, 57, 70, 96, 55, 93]

(A101s) [41, 71, 32, 91, 40, 84, 13, 64, 75, 0, 60, 25, 33, 82, 34, 52, 74, 70, 90, 42, 38, 54, 11, 58, 87, 22, 39, 95, 68, 37, 16, 18, 30, 49, 9, 69, 79, 86, 1, 61, 94, 83, 89, 7, 88, 76, 5, 98, 28, 15, 43, 56, 77, 6, 2, 31, 35, 81, 10, 4, 59, 19, 66, 36, 46, 48, 51, 45, 29, 72, 8, 55, 99, 12, 44, 17, 26, 78, 27, 67, 80, 92, 53, 20, 65, 3, 96, 14, 21, 85, 24, 97, 47, 50, 62, 57, 73, 23, 63, 93]

(A102i) [42, 31, 27, 39, 78, 48, 35, 23, 49, 96, 50, 16, 35, 34, 22, 93, 64, 50, 85, 69, 71, 72, 94, 63, 67, 6, 85, 82, 10, 8, 46, 65, 36, 80, 65, 83, 63, 24, 46, 73, 25, 93, 64, 76, 29, 72, 98, 93, 68, 92, 68, 91, 5, 92, 32, 16, 47, 35, 32, 10, 64, 63, 14, 62, 85, 47, 51, 21, 83, 74, 62, 0, 21, 14, 50, 14, 20, 51, 6, 60, 92, 82, 86, 31, 22, 90, 20, 81, 95, 78, 36, 33, 20, 51, 28, 91, 52, 75, 40, 3]

(A102s) [56, 18, 27, 39, 81, 48, 35, 23, 50, 98, 54, 1, 38, 34, 84, 93, 64, 26, 37, 69, 71, 7, 94, 11, 59, 57, 85, 82, 10, 8, 44, 41, 30, 80, 65, 19, 33, 24, 46, 73, 6, 79, 55, 76, 29, 72, 17, 96, 12, 0, 68, 91, 5, 92, 77, 16, 47, 2, 32, 42, 15, 63, 99, 87, 70, 45, 13, 89, 83, 97, 62, 66, 21, 14, 4, 25, 20, 88, 43, 60, 53, 74, 86, 31, 22, 90, 9, 49, 95, 78, 36, 58, 61, 51, 28, 67, 52, 75, 40, 3]

(A103i) [69, 96, 0, 90, 89, 87, 61, 21, 24, 86, 71, 34, 6, 78, 62, 5, 69, 78, 38, 0, 76, 72, 12, 68, 62, 36, 39, 3, 20, 2, 69, 74, 93, 14, 12, 10, 15, 68, 4, 23, 96, 84, 65, 99, 15, 41, 57, 24, 71, 49, 70, 2, 48, 64, 79, 22, 90, 26, 27, 52, 25, 7, 90, 45, 95, 13, 62, 54, 35, 79, 95, 24, 15, 4, 45, 48, 95, 3, 48, 89, 31, 29, 28, 65, 30, 24, 57, 56, 84, 42, 89, 99, 82, 58, 99, 56, 74, 11, 65, 45]

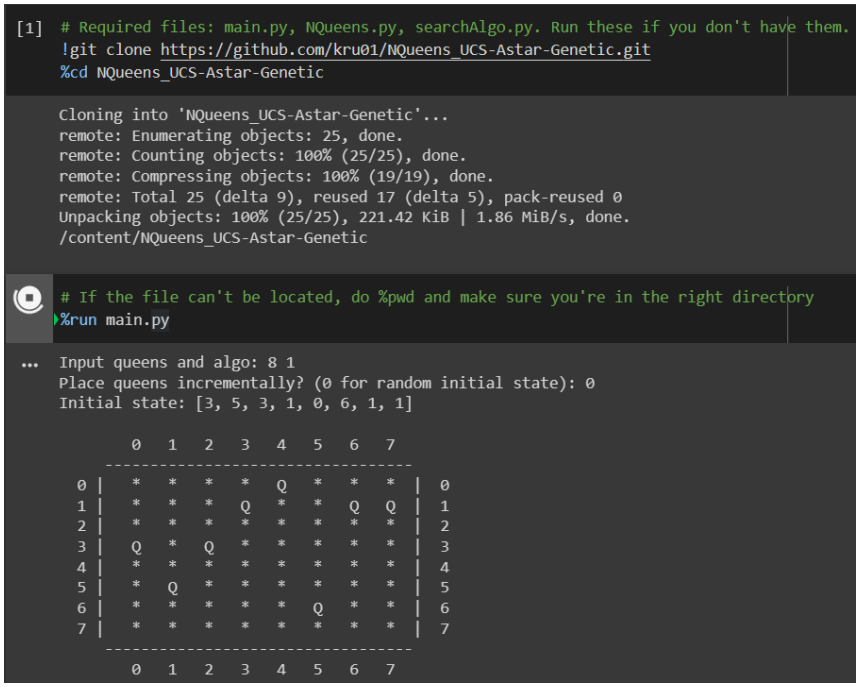
(A103s) [7, 96, 0, 9, 37, 87, 61, 21, 55, 86, 47, 77, 6, 22, 62, 85, 17, 19, 38, 51, 76, 72, 44, 63, 59, 97, 39, 91, 20, 53, 69, 43, 93, 14, 78, 10, 25, 68, 4, 23, 94, 98, 33, 46, 80, 5, 18, 26, 71, 49, 66, 2, 8, 64, 79, 73, 90, 45, 27, 52, 75, 88, 92, 67, 81, 12, 40, 54, 35, 32, 34, 1, 15, 83, 16, 48, 95, 3, 13, 41, 60, 29, 50, 65, 30, 24, 57, 28, 84, 42, 89, 36, 82, 70, 99, 56, 74, 11, 31, 58]

Genetic Algorithm

N	Run	Platform	Solution	Generation	Runtime (ms)	Memory (MB)
8	*1*	Colab	[6, 1, 5, 2, 0, 3, 7, 4]	119468	2283550.3097	0.5302
	2	Colab	[4, 6, 0, 2, 7, 5, 3, 1]	137155	1696469.0000	0.0322
	3	Colab	[4, 0, 3, 5, 7, 1, 6, 2]	154126	3846999.8956	0.8284

	Average		2609006.402	0.4636
100	*0*	Intractable		
500	*0*	Intractable		

4. IPython Notebook



I have chosen to tackle the single .ipynb file by not stuffing all of my codes in there since I don't like how it make the file all clustered and lengthy, it defeats my purpose of writing readable code. Instead, I offer one option to clone the github repository that I have created, if you don't have it yet, and one to execute the program once all the requirements are satisfied – all the necessary files are present, and the directory is correct.

The link for my repository I will also leave here in case you want to download the code without opening the notebook.

- https://github.com/kru01/NQueens_UCS-Astar-Genetic

5. Reference

UCS and A*

- **Slides provided by Dr. Nguyen Hai Minh.**
  - Lecture03-P1-ProblemSolvingBySearching
  - Lecture03-P2-UninformedSearch

    - Lecture03-P3-InformedSearch
- https://github.com/MManoah/NQueens-OptimalPathAlg

Genetic Algorithm

- **Slides provided by Dr. Nguyen Hai Minh.**
    - Lecture04-LocalSearch
- **An Adaptive Genetic Algorithm for Solving N-Queens Problem.**
    - https://arxiv.org/abs/1802.02006
    - https://arxiv.org/ftp/arxiv/papers/1802/1802.02006.pdf