

Faculty of Information Technology
University of Science, VNU HCMC

AI Lab 01 Report

Prepared by
Thien Huynh Duc - 21127693
July 03, 2023



Table of Contents

1. Project summary	2
1.1. Information	2
1.2. Introduction	2
2. Completion levels evaluation	2
3. Algorithm description	3
3.1. Uniform-cost search	3
3.2. A*	4
3.3. Genetic algorithm	4
4. Program	5
4.1. Supporting classes	5
4.2. Algorithm implementation	6
5. Statistics and charts	8
5.1. Statistics	8
5.2. Charts	11
6. Observations and Comparison	12
6.1. Observations	12
6.2. Comparison	13
Reference	13

1. Project summary

1.1. Information

Project

Search Strategies on N-queens problem

Student

Full name: Huynh Duc Thien

ID: 21127693

Contact: hdthien21@clc.fitus.edu.vn

Course: CSC14003 - Artificial Intelligence

Class: 21CLC07

Mentor

Teacher, Nguyen Ngoc Thao

Teacher, Le Ngoc Thanh

Teacher, Nguyen Tran Duy Minh

1.2. Introduction

N-queens problem is a well-known classic puzzle that involves placing **N chess queens** on an **NxN chess board** so that no two queens threaten each other (A queen attacks any piece in the same row, column, or diagonal). This problem, which remains a useful test problem for search algorithms, has been extensively studied in computer science and mathematics, and it has many applications in various fields such as artificial intelligence, operations research, and computer vision.

This project covers some features of applying search strategies and algorithms to solve the N-queens problem. The algorithms used in this project are **Uniform-cost Search (UCS)**, **A* search**, and **Genetic algorithm**, the performance will be judged by the running times and consumed memory with different numbers of queens in the problem. Due to the limited computational resources, some algorithms do not work efficiently. The project is managed to complete with admissible results.

2. Completion levels evaluation

No.	Work	Completion (%)
1	Algorithm design and implementation	
	<ul style="list-style-type: none"> Uniform-Cost Search (UCS) 	100

	• A*	100
	• Genetic algorithm	100
2	Simple console interface for input, output	100
3	Testing judgment	
	• N = 8	100
	• N = 100	33.33
	• N = 500	0
4	Running time and consumed memory statistics, charts	100
5	Comparison and observation	100

3. Algorithm description

3.1. Uniform-cost search

Uniform-cost search (UCS) is a search algorithm, used for finding the optimal path from a start node to a goal node in a graph or tree. It explores the search space uniformly by preferring the lower cumulative cost path. UCS uses a priority queue for storing nodes that have been explored, begins with a start node, keeps generating neighboring nodes, does the expanding and updating costs accordingly, and stop when it reaches a goal node or no available state remains.

Pseudo code for uniform-cost search applied in this project:

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /*chooses the lowest-cost node in
frontier*/
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then

```

replace that *frontier* node with *child*

3.2. A*

A* is a search algorithm used to find the optimal path between a start node and a goal node in a graph or tree. The priority of each node in the A* algorithm is determined by a combination of cumulative cost path and heuristic estimated by a special function. The algorithm starts at the initial node and expands the node with the lowest combined cost first, generating all its neighbors and calculating their combined costs. The algorithm continues to expand nodes with the lowest combined cost until it reaches the goal node.

Pseudo code for A* algorithm applied in this project:

```

function A-STAR-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST + node.HEURISTIC
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /*chooses the lowest F node in frontier*/
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      child.F ← child.PATH-COST + child.HEURISTIC
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher F then
        replace that frontier node with child
  
```

3.3. Genetic algorithm

Genetic algorithm is a type of optimization algorithm that emulates the process of natural selection and genetics to find approximate solutions to optimization and search problems. It starts with a set of initial states, also called population, of potential solutions for applying genetic operators such as random selection, crossover, and mutation to evolve the population toward the best solutions. The algorithm uses a fitness function, and probability to evaluate the quality of each individual and select the best ones for propagating to future generations. Genetic algorithms are usually applied in solving complex or large search spaces and can converge on high-quality solutions over time.

Pseudocode for Genetic Algorithms applied in this project:

```

function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  input: population, a set of individuals
           FITNESS-FN, a function measuring the fitness of an individual
  repeat
    new_population  $\leftarrow$  empty set
    for i = 1 to SIZE(population) do
      x  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      y  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      child1, child2  $\leftarrow$  Crossover(x, y)
      if (small random probability) then
        child1  $\leftarrow$  MUTATE(child1)
        child2  $\leftarrow$  MUTATE(child2)
      add child1, child2 to new_population
  until some individual fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

```

4. Program

4.1. Supporting classes

- **NQueens** class gives an overview representation of the N-Queens problem, the state is a list of integers, and each value is ranged from 0 to n-1 to identify the position of the queen in that column.
- **Priority Queue** data structure used in UCS and A* algorithm is a reconstruction of the built-in library **heapq** for specific features of this program, which helps to ensure data is always arranged in the right order (ascending order). Due to this mechanism, the performance of the program can be improved by reducing the amount of time searching for the most-prior state.
- **Vertex** is the smallest unit defined in this program, which is a combination of state, path_cost, heuristic, f_value, and n, belonging to the state of the chess board. The class provides several functions for calculating path_cost (g), heuristic (h), f_value (f), and logical methods supporting finding the problem's solution. **UcsVertex** and **AstarVertex** are two subclasses characterized for different algorithms UCS and A*. They contain `__lt__()`, which defines the comparing mechanism in the two above algorithms.
- **GeneticAlg** class implements the necessary attributes and methods for the genetic algorithm, including `generate_population`, `random_selection`,

crossover, and random_mutation, simulating the natural selection process.

4.2. Algorithm implementation

- UCS

```
def uniform_cost_search(initial_vertex: UcsVertex):
    frontier = PriorityQueue()
    frontier.enqueue(initial_vertex)
    explored = []
    while not frontier.is_empty():
        ucs_vert = frontier.dequeue()
        if ucs_vert.is_goal():
            return ucs_vert.state
        explored.append(ucs_vert.state.copy())
        for s_vert in ucs_vert.generate_successors():
            frontier.enqueue(s_vert)
    print('Solving fail.')
    return None
```

In this implemented UCS algorithm, first of all, a priority queue **frontier** is created for storing all the vertices for expanding according to the ascending order of **path_cost**. The smallest path_cost vertex will be popped out for expanding continuously. If the vertex contains the goal state, then the result will be returned. Otherwise, the state of that vertex will be checked if it has already been in **explored** set (which is a list of explored states). In the case it is, another vertex will be considered, if not, its successors will be included in the frontier for future exploration, and the current state will be added to the explored set. In the end, if the frontier comes empty without any solution, **None** will be returned.

- A*

```
def a_star_search(initial_vertex: AstarVertex):
    frontier = PriorityQueue()
    frontier.enqueue((initial_vertex.heuristic, 0, initial_vertex.state))
    explored = []
    while not frontier.is_empty():
        heuristic, path_cost, state = frontier.dequeue()
        as_vert = AstarVertex(state, path_cost, heuristic)
        if heuristic == 0:
            return state
        explored.append(state)
        for s in as_vert.generate_successors(state):
            if s not in explored:
                frontier.enqueue((as_vert.h(s), path_cost + 1, s))
    print('Solving fail.')
    return None
```

The A* algorithm starts by creating a priority queue **frontier** for tracking created vertex but has not yet been explored. The queue is sorted in the ascending order of **f_value**, which is the sum of **path_cost** and **heuristic** cost of each state contained in the vertices. For each turn of the algorithm, the least priority value vertex will be considered. The solution will be returned in case the program encounters a vertex with 0 heuristic cost (which represents the goal state). On the contrary, the vertex's successors will be generated. For all the successors that have been created, the already explored ones will be discarded, and the others will be added in order of future explorations. In such a situation, all the possible vertices have been expanded without an outcome solution, the **None** value is returned.

- Genetic algorithm

```
def genetic_algorithm(initial_state: list):
    gen_alg = GeneticAlg(initial_state, len(initial_state))

    pop_size = len(initial_state)
    population = gen_alg.generate_population(pop_size)

    solution = gen_alg.find_goal(population)

    while solution is None:
        new_population = []
        for _ in range(int(pop_size/2)):
            x, y = gen_alg.random_selection(population)

            # generate new generation
            c1, c2 = gen_alg.crossover(x, y)

            if gen_alg.probability(c1) < 0.2:
                gen_alg.random_mutation(c1)
            if gen_alg.probability(c2) < 0.2:
                gen_alg.random_mutation(c2)

            new_population.append(c1)
            new_population.append(c2)

        del population
        population = new_population

        solution = gen_alg.find_goal(population)

    return solution.state
```

At first, an initial population is created for evolution. All the created vertex will be examined for containing any goal state, the found solution is returned. If not, then, each random selection of two individuals is applied for generating two new children individuals through the **crossover** method.

The ones with a small probability of fitness are chosen for mutation, creating a new child. The **find_goal** method is carried out, seeking the goal state in the **new_population**. Each turn the old population will be replaced with the new population, and the program runs continuously until a solution is found.

5. Statistics and charts

5.1. Statistics

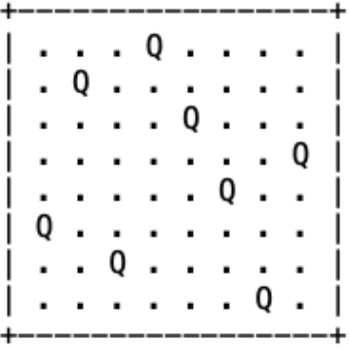
Alg	Running time (ms)			Memory (MB)		
	N=8	N=100	N=500	N=8	N=100	N=500
UCS	6329.21	Intractable	Intractable	0.01221	Intractable	Intractable
A*	12.1322	1481552.2	Intractable	0.01528	0.133896	Intractable
GA	15105.1	Intractable	Intractable	0.00386	Intractable	Intractable

Average running time and memory usage of 5 test cases

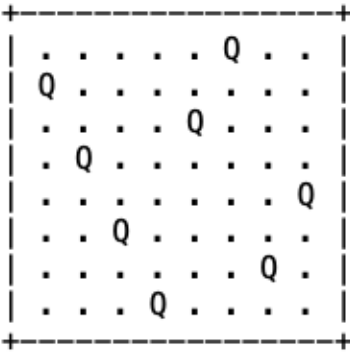
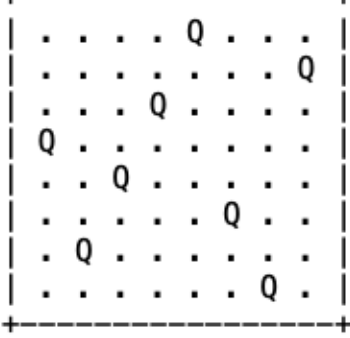
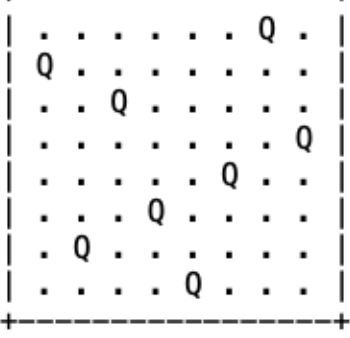
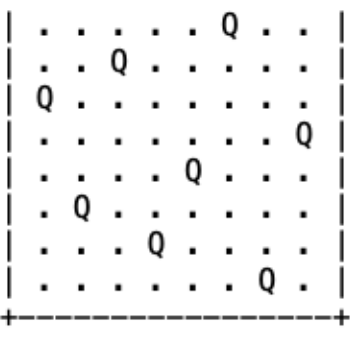
Computer configuration applied for running the program:

- 2.80 GHz CPU (4 cores, 8 threads)
- 16GB RAM 3200 MHz (10GB available)
- 3500 MB/s SSD

For some of N=100 and N=500 cases, the appropriate solutions are not findable due to the large time consumptions (over 12 hours of elapsed time for each test case) and temporary memory runs out of space.

	Initial State	Solution State	Solution Board
UCS	[5, 6, 2, 0, 3, 4, 7, 1]	[5, 1, 6, 0, 2, 4, 7, 3]	

	[7, 4, 0, 5, 1, 6, 3, 2]	[7, 2, 0, 5, 1, 4, 6, 3]	
	[1, 0, 7, 6, 4, 5, 2, 3]	[2, 5, 7, 0, 4, 6, 1, 3]	
A*	[3, 7, 1, 2, 0, 5, 6, 4]	[3, 7, 0, 2, 5, 1, 6, 4]	
	[4, 1, 3, 5, 7, 6, 2, 0]	[4, 1, 3, 5, 7, 2, 0, 6]	

	[5, 3, 2, 7, 1, 0, 6, 4]	[1, 3, 5, 7, 2, 0, 6, 4]	 <p>An 8x8 grid with 8 Queens placed at (row, col): (1,8), (2,1), (3,6), (4,7), (5,3), (6,5), (7,2), (8,4). Rows and columns are indexed 1 to 8 from top-left.</p>
GA	[3, 0, 6, 2, 1, 5, 7, 4]	[3, 6, 4, 2, 0, 5, 7, 1]	 <p>An 8x8 grid with 8 Queens placed at (row, col): (1,6), (2,8), (3,4), (4,1), (5,3), (6,7), (7,5), (8,2). Rows and columns are indexed 1 to 8 from top-left.</p>
	[1, 7, 2, 5, 0, 6, 3, 4]	[1, 6, 2, 5, 7, 4, 0, 3]	 <p>An 8x8 grid with 8 Queens placed at (row, col): (1,8), (2,1), (3,6), (4,7), (5,3), (6,5), (7,2), (8,4). Rows and columns are indexed 1 to 8 from top-left.</p>
	[7, 5, 3, 1, 4, 0, 6, 2]	[2, 5, 1, 6, 4, 0, 7, 3]	 <p>An 8x8 grid with 8 Queens placed at (row, col): (1,6), (2,8), (3,4), (4,1), (5,3), (6,7), (7,5), (8,2). Rows and columns are indexed 1 to 8 from top-left.</p>

Sample solution for each algorithm

5.2. Charts

In order to analyze the experimental results, the following charts illustrate the relationship between running times, memory usage, and N . Each chart is based on the outputs of 10 test cases.

- With $N=8$

Detail view:  ucs_n8.png

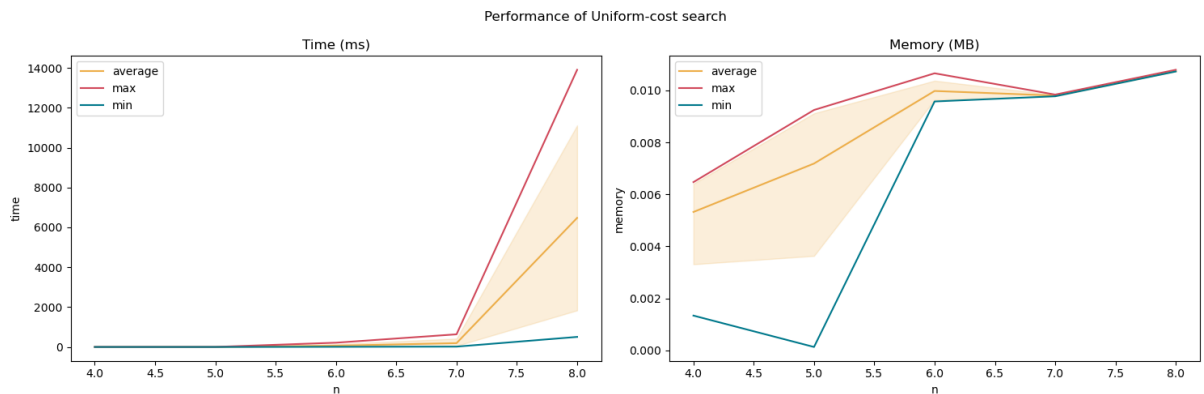


Figure 1: Performance of UCS chart ($N=8$)

Detail view:  astar_n8.png

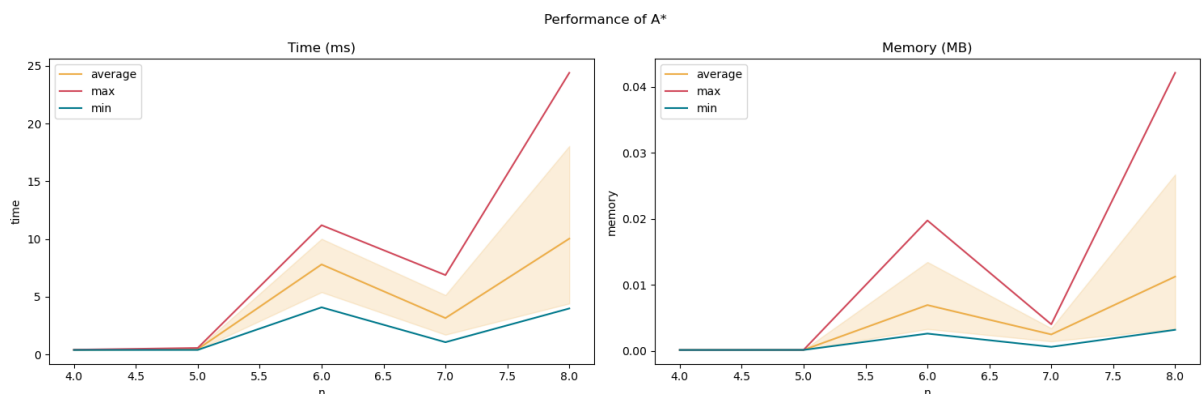



Figure 2: Performance of A* chart ($N=8$)

Detail view:  genetic_algorithm_n8.png

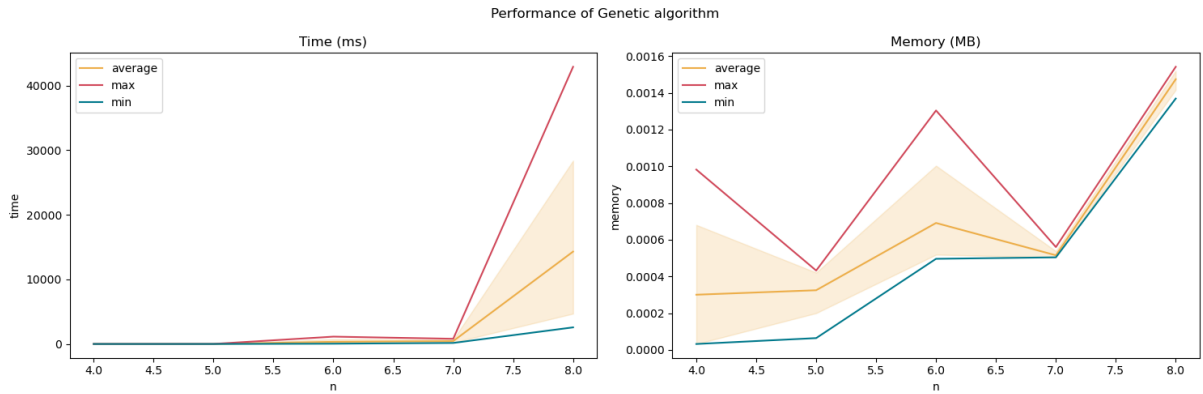


Figure 3: Performance of GA chart (N=8)

- With **N=100**

Detail view:  astar_n100.png

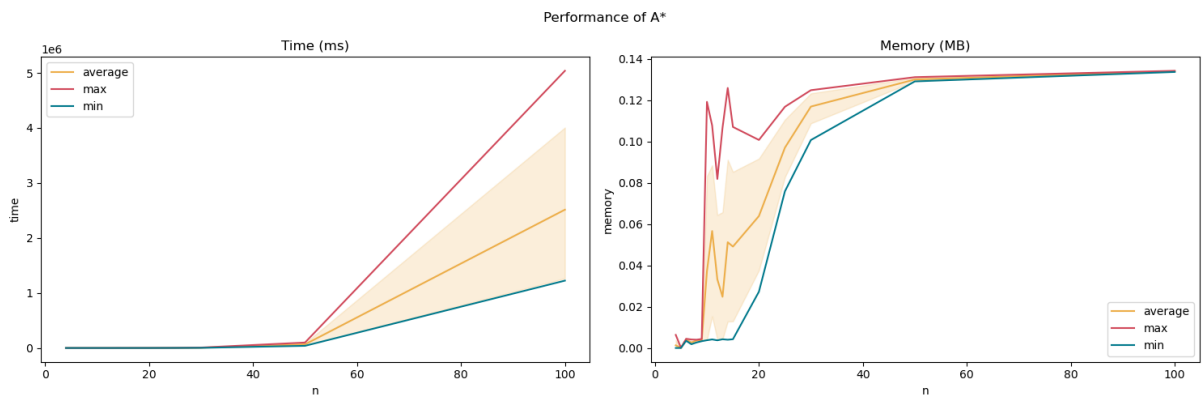


Figure 4: Performance of A* chart (N=100)

In general, as the value of n increases, both the time and memory usage tend to increase. However, for each specific case, there are still different random variations.

6. Observations and Comparison

6.1. Observations

For the N-Queens problem, the UCS algorithm does not guarantee its characteristic property where, for the same successor generation of a vertex, the path cost is precisely the same. The priority is not explicitly stated, and at this point, the algorithm is quite similar to BFS. Consequently, the search space and search time are both significant.

Similarly, for the A* algorithm, it becomes relatively similar to GBFS when the heuristic value plays a crucial role in guiding the search, as the significance of the path cost is not substantial. As a result, the search efficiency of the algorithm is somewhat reduced.

6.2. Comparison

UCS and A* algorithms offer efficient and exact solutions for the N-Queens problem with the small size of the problem (N). However, as the problem size increases, these algorithms face challenges due to the significantly increased number of states to explore, resulting in slower execution time and difficulty in reaching the goal state.

In contrast, the Genetic Algorithm (GA) can provide precise solutions for the problem when an appropriate number of individuals is utilized. However, GA generally requires more time to find the goal compared to UCS and A* algorithms.

To summarize, for small N-Queens problems, it is advisable to employ UCS and A* algorithms for fast results. However, when dealing with larger problem sizes or when a longer execution time is acceptable, the Genetic Algorithm is a suitable choice.

Reference

Pseudo code and idea implementation for three algorithms: *Stuart Russell and Peter Norvig. Artificial Intelligence: A Modern Approach (4th ed.).*

Better observation of NQueens Problem: *Materials (slides and exercises) of Artificial Intelligence course - CSC14003 - Teacher Nguyen Ngoc Thao*

–The end–