VIETNAM GENERAL CONFEDERATION OF LABOR

TON DUC THANG UNIVERSITY

FACULTY OF INFORMATION TECHNOLOGY

Group 4

Introduction to Artificial Intelligence

Presentation

HO CHI MINH CITY, 2025

TABLE OF CONTENT

STUDENT LISTS

Student ID	Full name	Email	Assigned tasks	Complete percentage
519K0078	Benedict Timothy Chibuike	519K0078@student.tdtu.edu.vn	Task 2 + Task 4	98%
523K0047	Trần Thị Thế Nhân	523K0047@student.tdtu.edu.vn	Task 3 + Task 4, presentation	100%
519C0006	Nguyễn Thị Quế Châu	519C0006@student.tdtu.edu.vn	Task 1 + Task 4	100%
520K0250	Nguyễn Tường Hoàng	520K0250@student.tdtu.edu.vn	Not attended	0%

TASK 1

1. Approaches to Solve Tasks (Using Pseudo Code & Diagrams)

A* Algorithm for 8-Puzzle Pathfinding:

• Initialize priority queue with the starting state.

• Use heuristic (Misplaced Tiles or Manhattan Distance) to estimate cost.

• Expand node with the lowest cost (f = g + h).

• If the goal is reached, return the path.

• Otherwise, explore neighbors, apply swap rules (1<->3, 2<->4), and update costs.

Pseudo Code:

```
function A_Star_Search(start, goal):
    frontier = priority_queue()
    explored = set()
    frontier.push((start, heuristic(start)))

    while frontier is not empty:
        current_node = frontier.pop()
        if goal_reached(current_node):
            return reconstruct_path(current_node)

        for neighbor in get_neighbors(current_node):
            apply_swap_rules(neighbor)
            if neighbor not in explored:
                frontier.push((neighbor, cost + heuristic(neighbor)))
                explored.add(neighbor)

    return failure
```

2. Avoiding Raw Source Code

Used clean pseudo code to demonstrate the logic of the A* search.

Focused on algorithm structure and high-level explanation, not implementation details.

3. Study Topics & Practical Examples

8-Puzzle Representation:

• A 3x3 grid with 8 tiles and 1 blank (0).

• Goal states: 4 valid configurations defined.

• After each move, check and auto-swap 1↔3 or 2↔4 if adjacent.

Heuristic Functions:

• Misplaced Tiles: Counts how many tiles are in wrong positions.

• Manhattan Distance: Sum of row and column distances to goal positions.

4. Advantages vs Disadvantages

• Misplaced Tiles:

- Advantage: Fast and simple.
- Disadvantage: Less accurate, may expand more nodes.

• Manhattan Distance:

- Advantage: More accurate heuristic, leads to shorter paths.
- Disadvantage: Slightly higher computation cost.

5. Completion Percentages

Feature	Completion
A* Algorithm	100%
Swap Logic for 1-3 and 2-4	100%
Heuristic Functions	100%
Experiment & Average Cost	100%
Visualization (Graphviz)	100%
Result Comparison Chart	100%

6. Code Explanation

1. Class: PuzzleState

Represents the current state of the 8-puzzle board.

• __init__(self, initial): Initializes the puzzle with a list of 9 tiles, stores the position of the blank (0).

• is_goal(self, goal_state): Checks whether the current state matches the goal state.

• get_neighbors(self): Returns a list of neighboring states by moving the blank tile. Also applies the special rule: auto-swap tile 1 and 3, or 2 and 4 if adjacent.

• print_state(self): Prints the puzzle in 3x3 format.

• __eq__(self, other): Compares two PuzzleState objects for equality.

• __hash__(self): Allows PuzzleState to be used in sets/dictionaries.

• is_solvable(state, goal_states): Static method to check if the puzzle can reach one of the goal states based on inversion count and blank tile row.

2. Class: Node

Represents a node in the A* search tree.

• __init__(self, puzzle, g, h, parent=None, move=None): Stores puzzle state, g-cost, h-cost, f-cost, parent, and move that led to this node.

• get_f(self): Returns f = g + h, used for priority in A*.

• __lt__(self, other): Allows nodes to be compared by f-cost for the priority queue.

3. Class: AStarSolver

Solves the puzzle using the A* algorithm and a chosen heuristic.

• __init__(self, heuristic): Stores the heuristic function.

• solve(self, start_puzzle, current_goal): Runs A* to find the shortest path to the goal. Maintains open and closed sets, expands the lowest f-cost node each time.

4. Heuristic Functions

• misplaced_tiles_heuristic(puzzle, goal_state): Counts tiles that are not in their goal position.

• manhattan_distance_heuristic(puzzle, goal_state): Calculates the total distance each tile is from its goal (row + column difference).

5. Class: PuzzleExperiment

Runs multiple random puzzle tests to evaluate heuristics.

• __init__(self, numtrials): Sets the number of trials.

• random_puzzle(self): Returns a random PuzzleState.

• run_experiment(self): Solves random puzzles using both heuristics, computes and prints average moves.

6. Class: SearchTreeVisualizer

Visualizes the search tree in text and graph format.

• puzzle_to_multiline(self, state): Converts a state into multi-line string (3x3 grid).

• illustrate_search_tree(self, start_puzzle, n): Prints a text-based tree with up to n node expansions.

• puzzle_to_str(self, state): Converts puzzle state to single string.

• illustrate_tree_graph(self, start_puzzle, n): Uses Graphviz to draw the search tree.

7. Main Function & Chart

• main(): User interface to input or randomize a puzzle, visualize the tree, solve with both heuristics, and run experiments.

• plot_comparison_chart(results): Displays a bar chart comparing average path costs of the two heuristics.

TASK 2

1. Approaches to Solve Tasks (Using Pseudo Code & Diagrams)

*A Algorithm for Pathfinding in Pac-Man Maze:**

Initialize priority queue with the starting position.

Use a heuristic function (Manhattan distance) to estimate cost.

Expand the node with the lowest cost.

If the goal (all food collected) is reached, return the path.

Otherwise, explore neighbors and update costs.

Pseudo Code:

```
function A_Star_Search(start, food_positions):
    frontier = priority_queue()
    explored = set()
    frontier.push((start, heuristic(start)))

    while frontier is not empty:
        current_node = frontier.pop()
        if goal_reached(current_node):
            return reconstruct_path(current_node)

        for neighbor in get_neighbors(current_node):
            if neighbor not in explored:
                frontier.push((neighbor, cost + heuristic(neighbor)))
                explored.add(neighbor)


    return failure
```


pygame window

2. Avoiding Raw Source Code

- Used pseudo code to represent A* algorithm logic.
- Explained core logic with practical steps rather than direct implementation.

---

3. Study Topics & Practical Examples

Maze Representation:

A grid-based layout where:

- '%' represents walls.
- 'P' represents Pac-Man.
- '.' represents food.
- 'O' represents pie (speed boost).

Teleportation Mechanism:

Four corners of the maze act as teleport portals, allowing Pac-Man to traverse quickly.

Visualization Using Pygame:

- Dynamic rendering of the Pac-Man environment with animation.
- Pie and food items appear with visual effects.
- Real-time movements following computed path.

4. Advantages vs Disadvantages

Approach	Advantages	Disadvantages
A Search*	Efficient and finds the shortest path	Can be slow for large maps
Teleportation	Allows quick traversal	Can complicate pathfinding logic
Pie Power-Up	Temporary invincibility	Needs careful tracking of state
Pygame Visualization	Enhances understanding through animation	Requires additional setup

5. Completion Percentages for Each Task

Task	Completion %
Implementing A* Algorithm	100%
Handling Teleportation	100%
Integrating Pie Power-up	100%
Visualization with Pygame	90% (Minor enhancements possible)
Maze Parsing and Execution	100%

TASK 3 : Solving the 16-Queens Problem

1. Approach to Solving using a Genetic Algorithm

Main Solution Flow

- Initialize population with random permutations
- For each generation:

  Calculate fitness for all individuals (count diagonal conflicts)

  If best fitness == 1.0 (no conflicts):

    Return solution

  Else:

    Evolve population:

      Select parents via tournament selection

      Perform ordered crossover (OX) to create offspring

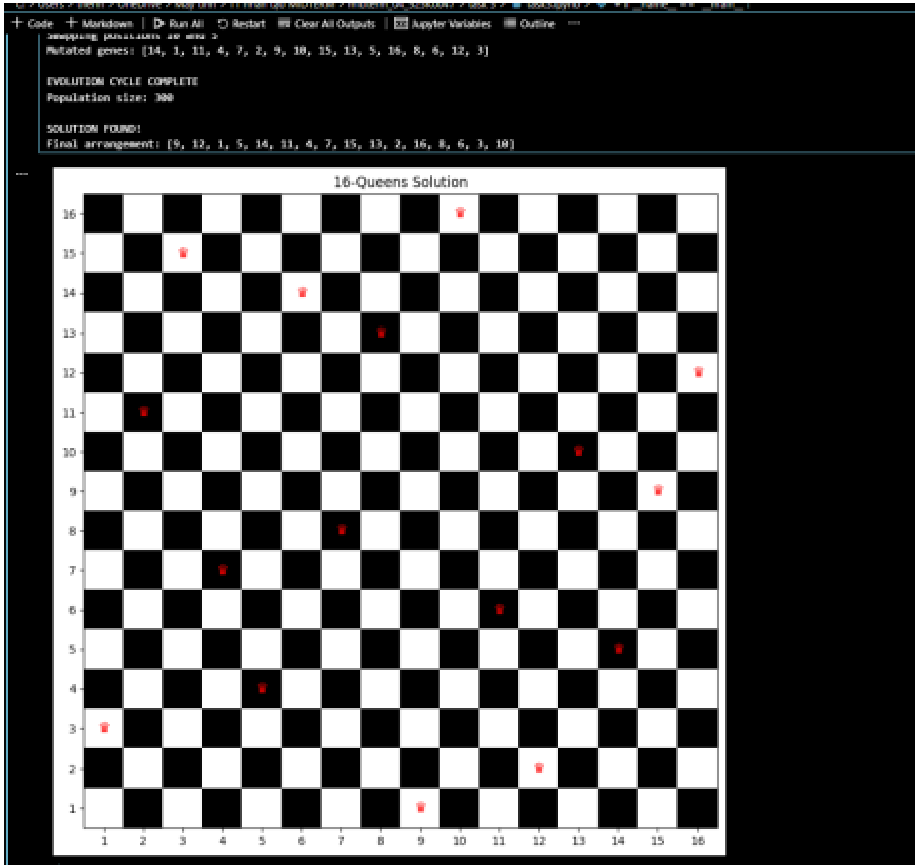
      Apply swap mutation to offspring

      Preserve top 5% elites

      Replace old population with new offspring + elites

- Repeat until solution found or max generations reached (until termination)

Visualization
i used matplotlib to display the chessboard with queens.



Example visualize of some approach
- Crossover (OX):

Parent1: [A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P]
Parent2: [Q|R|S|T|U|V|W|X|Y|Z|α|β|γ|δ|ε|ζ]
Crossover points: 3 and 9 → Child: [Q|R|S|D|E|F|G|H|I|T|U|V|W|X|Y|Z].
- Mutation: Swap positions 4 and 15 → [1, 2, 3, 4, 16, ..., 5].

My improvements from the first attempt
- Ordered Crossover (OX): Replaced two-point crossover at the first time to preserve valid permutations. For example: Child inherits a segment from Parent1 and fills gaps from Parent2.
- Select parents via Tournament selection instead of Rank-based selection due to simplicity and speed.
- Elitism: Preserves top 5% of solutions, (previously fixed 2 elites: in evolve() function, elites = self.population[:2])
- Parameters: Try larger population (300) and generations (2000) improve success rate, first time I used 100 population and 1000 generations.

2. Advantages vs. Disadvantages

Advantages	**Disadvantages**
Handles large search spaces (16! permutations).	No guaranteed convergence (may fail randomly).
Parallel exploration of solutions.	Requires parameter tuning (e.g., mutation rate).
Flexible representation (permutation encoding).	Computationally expensive for large populations.

3. Study Topics

Topic	Description	Example
Genetic Algorithms	Evolutionary optimization using selection, crossover, and mutation.	Solving 16-Queens as a permutation problem.
Permutation Encoding	Representing solutions as unique sequences (e.g., columns for queens).	genes = [2, 5, 11, ...] → no row conflicts.
Fitness Function	Quantifying solution quality (e.g., inverse of diagonal conflicts).	2 conflicts → Fitness = 1/3 ≈ 0.333.
Local Search	Iteratively improving solutions through neighborhood exploration.	Mutation swaps two genes to create neighbors.

4. Task Completion Table

Task Requirement	Implementation Percent	Code Reference	Notes
Formulate as a local search problem	100%	Individual class: genes and calculate_fitness()	Diagonal conflicts checked via absolute differences.
Implement genetic algorithm	100%	GeneticAlgorithm class: _select_parents(), _crossover(), _mutate(), evolve()	Uses tournament selection and ordered crossover (OX).
Crossover at two random points	100%	_crossover(): point1, point2 = sorted(random.sample(range(16), 2))	Modified to use OX for validity.
Flexible mutation rate	100%	__init__(): mutation_rate=0.15 _mutate(): Applies swap if random.random() < mutation_rate	Rate adjustable during initialization.
State representation	100%	Individual.__init__(): self.genes = random.sample(range(1, 17), 16)	Ensures no row/column conflicts.
Successor function	100%	_mutate(): Swaps two genes _crossover(): Combines parent segments	Mutation explores local neighbors; crossover explores global combinations.
Fitness function	100%	calculate_fitness(): Counts diagonal conflicts via abs(i-j) == abs(genes[i]-genes[j])	Normalized to (0, 1] range.
OOP design (compact/reasonable)	100%	Entire code structure	Methods are logically grouped.
Visualization	100%	visualize_solution(): Uses matplotlib to display queens on a 16x16 grid	Red queens mark positions.
User-defined parameters	100%	GeneticAlgorithm(population_size=300, mutation_rate=0.15) run(max_generations=2000)	Defaults provided but customizable.
Termination condition	100%	run(): Checks if best.fitness == 1.0	Max generations increased to 2000 for better convergence.