INTRODUCTION TO ARTIFICIAL INTELLIGENCE

MIDTERM PRESENTATION

I. INTRODUCTION

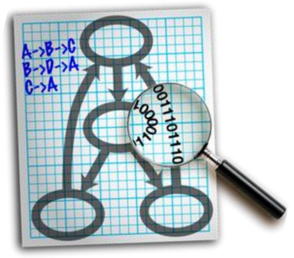
Student ID	Fullname	Assign tasks	Complete Percentage
523H0115	Nguyễn Trần Nhật An 523H0115@student.tdtu.edu.vn	Task 1	100%
523H0185	Nguyễn Phúc Toàn 523H0185@student.tdtu.edu.vn	Task 2	100%
523H0140	Nguyễn Quang Huy 523H0140@student.tdtu.edu.vn	Task 3	100%
523H0196	Chung Quang Vũ 523H0196@student.tdtu.edu.vn	Task 4	100%
523H0187	Nguyễn Minh Trí 523H0187@student.tdtu.edu.vn	Task 4	100%

II. IMPLEMENTATION:

Task 1

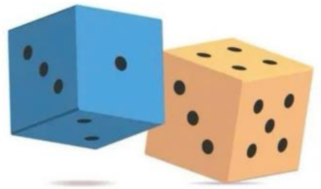
Solving 8-Puzzle using A*

Essential libraries and modules



GRAPHVIZ

Generates visualizations of the search tree and solution path



RANDOM

Shuffles tiles to generate random initial states

Essential libraries and modules

HEAPQ

Implements a priority queue (min-heap) for efficient state expansion in the A* search.

COPY

Creates independent copies of puzzle states during transitions to avoid reference conflicts.

Class Puzzle

- Represents a single state of the 8-puzzle board and manages the state transitions within the search algorithm

- Attributes:

  - **state:** 3x3 grid
  - **id:** unique string representation of state
  - **action:** move taken to reach this state
  - **parent:** previous state

Class Puzzle

- Represents a single state of the 8-puzzle board and manages the state transitions within the search algorithm

- Attributes:

  - **g** = cost since intitial state
  - **h** = estimated cost to goal state
  - **f = g + h**

Class Puzzle

- Essential methods:

- **get_pos(state, value):**

```
for each row and column in state:
    if value found: return (row,col)
```

- **swap(a, b):**

```
get positions of a and b
swap their values
```

Class Puzzle

- Essential methods:

  - **check_neighbor(state, a, b):**

```
get positions of a and b
        return true if adjacent
```

Class Puzzle

- Essential methods:

  - **get_dest_pos(action, pi, pj):**

    if action is 'L': return (pi, pj + 1)

    if action is 'R': return (pi, pj - 1)

    if action is 'U': return (pi + 1, pj)

    if action is 'D': return (pi - 1, pj)

Class Puzzle

- Essential methods:

  - **get_successor(action, state):**

```
find position of empty tile (0)
get new position based on action
if new position is valid:
        swap values
        return new state
else return None
```

Class Puzzle

- Essential methods:

  - **get_successors**():

```
check if 1-3 and 2-4 are neighbors initially
successors = []
for each possible move ('L', 'R', 'U', 'D'):
        generate new state
```

Class Puzzle

- Essential methods:

  - **get_successors**():

```
if new_state is valid:
    if 1-3 are now neighbors but weren't before:
        swap 1 and 3
  if 2-4 are now neighbors but weren't before:
        swap 2 and 4
 add new Puzzle state to successors
```

Class Puzzle

- Essential methods:

  - **get_solution_path**():

```
path = []
node = current node
    while node has parent:
            add node's action to path
            node = node.parent
    return reversed path
```

Class Puzzle

- Essential methods:

- **draw(dot):**

```
generate a table representation
add node to dot
if parent exists:
        create edge between parent and current node
```

Class PuzzleAgent

- Implements an A* search algorithm to solve the 8-puzzle, that supports multiple goal states, heuristics functions, and visualization.

- Attributes:

  - **dot graph for visualization**
  - **explored** = set()
  - **drawn** = set()
  - **open_set** = []
  - **state_map**(maps state string to Puzzle object)

Class PuzzleAgent

- Essential methods:

- **solve**():

  + Compute initial heuristic (h0) to any goal state

  + Create start Puzzle node with g=0, h=h0

  + Add start node to open_set and state_map

Class PuzzleAgent

- Essential methods:

  - **solve()**:

```
while open_set is not empty:
        extract node with lowest f from open_set
        if node's state matches any goal state:
            if node not drawn:
                draw node and add to drawn
            node = node.parent
```

Class PuzzleAgent

- Essential methods:

  - **solve()**:

```
for each successor of current node:
    if successor already explored, continue
    compute g, h, and f values for successor
    if successor not in state_map or has better f value:
        add to open_set
        update state_map
```

Heuristic functions

- **h_manhattan(state, goal):**

```
distance = 0
for each tile 1-8:
    get positions in state and goal
    compute Manhattan distance
return distance
```

Heuristic functions

- **h_near_goal(state, goal, n=2):**

```
count = 0
for each tile 1-8:
    get positions in state and goal
    if Manhattan distance ≤ n:
        count +=1
return count
```

Heuristic functions

Heuristic function	**Admissibility**	**Consistency**
h_manhattan (Sum of Manhattan distances for all tiles)	- Never overestimates the true cost to the goal. - Each tile must move at least its Manhattan distance to reach its goal position. - Summing these distances gives a lower bound on the total moves needed.	- For any move, the heuristic function satisfies: $\mathbf{h(n) \leq cost(n\square\ n\square)+h(n\square)}$ - Ensures A* finds the optimal path without reopening nodes.

Heuristic functions

Heuristic function	**Admissibility**	**Consistency**
h_near_goal (Counts tiles within n=2 moves of their goal)	- Counts how many tiles are "close enough" to their goal positions. - Since it only considers tiles within a small distance (n=2), it never overestimates the true cost.	- Similar logic applies: moving a tile can change its status at most 1, so it satisfies: $h(n) \leq 1 + h(n')$

Main execution

```
goal_states = list of goals
generate random initial_state
result, graph = PuzzleAgent.solve(initial_state, goal_states, h_manhattan)
if result:
        print solution information
        display graph
   else:
        no solution found
```

Evaluation table

Aspect	Advantages	Disadvantages	Completion Status
*Algorithm (A)**	Guarantees optimal solution with admissible and consistent heuristics.	Memory-intensive (stores all explored states).	100%
Special Rules	Correctly handles unique constraints	Increases complexity of successor generation.	100%
Visualization	Helpful visual debugging tools	Limited by graph_depth	100%
Heuristics	Supports customizable heuristics	Requires careful heuristic design for best results	100%
Multiple Goals	Solves for multiple goal states in one run.	Computes heuristics for all goals, adding overhead.	100%

II. IMPLEMENTATION:

Task 2

Pathfinding Algorithm for Pac-Man Navigation using A*

Essential libraries and modules

HEAPQ

Implements a priority queue (min-heap) for efficient state expansion in the A* search.

PYGAME

Creates a graphical user interface for rendering, displaying sprites, and updating the game state on screen...

Essential libraries and modules

ITERTOOLS

Compresses a sequence of moves by grouping consecutive identical directions.

Essential constants and parameters

Pos = tuple[int, int]	A type alias for position coordinates.
Directions{}	Map direction names to coordinate offsets.

Game Background

- You are **Wilbur** the goldfish and your job is to collect all **pearls** scattered across the map.
- On each corner, there is a **portal**. When landing on it Wilbur will get teleported to the opposite corner (e.g. Top left ⍰ Bottom right).
- There are also **gems** – magical collectibles that allow Wilbur to “ghost” through walls for 5 turns.

Class Game

- Responsible for storing a state of the game and handling the rules.

- Attributes:

  - player: The current position of Wilbur.
  - pearls: A set of uncollected pearls.
  - gems: A set of uncollected gems.
  - ghost_turns: Number of turns left for ghost mode.
  - walls: A set of wall coordinates.
  - portals: Teleporting locations.

Class Game

- Essential methods:

  - **load_map(map_str)**: Creates a game state from a map string.

```
width, height = length rows and columns
FOR each character in map_str:
        IF “P”: SET player_pos
        IF “.”: ADD to pearls
        IF “O”: ADD to gems
        IF “%”: ADD to walls
```

Class Game

- Essential methods:

  - **get_moves()**: Determines all valid moves at the current state.

```
x, y = player
moves = {}
FOR each direction(dx, dy) in directions:
    new_pos = (x+dx, y + dy)
    IF new_pos is OUTSIDE map boundaries: CONTINUE
    IF new_pos is a wall AND ghost_turns == 0: CONTINUE
    IF new_pos is a portal: new_pos = TELEPORTING_POSITION
ADD {direction: new_pos} to moves
```

Class Game

- Essential methods:

- **move_to(new_pos)**: Creates a new state after moving to a new position.

```
IF new_pos == current position:  # No need to move
        RETURN self
new_pearls = COPY(self.pearls)
new_gems = COPY(self.gems)
new_ghost_turns = self.ghost_turns - 1
```

Class Game

- Essential methods:

- **move_to(new_pos)**: (cont.)

```
IF new_pos is in new_pearls:
    REMOVE new_pearls(new_pos)
IF new_pos is in new_gems:
    REMOVE new_gems(new_pos)
    new_ghost_turns = 5
RETURN Game(new_pos, new_pearls, new_gems, new_ghost_turns)
```

Class Game

- Essential methods:

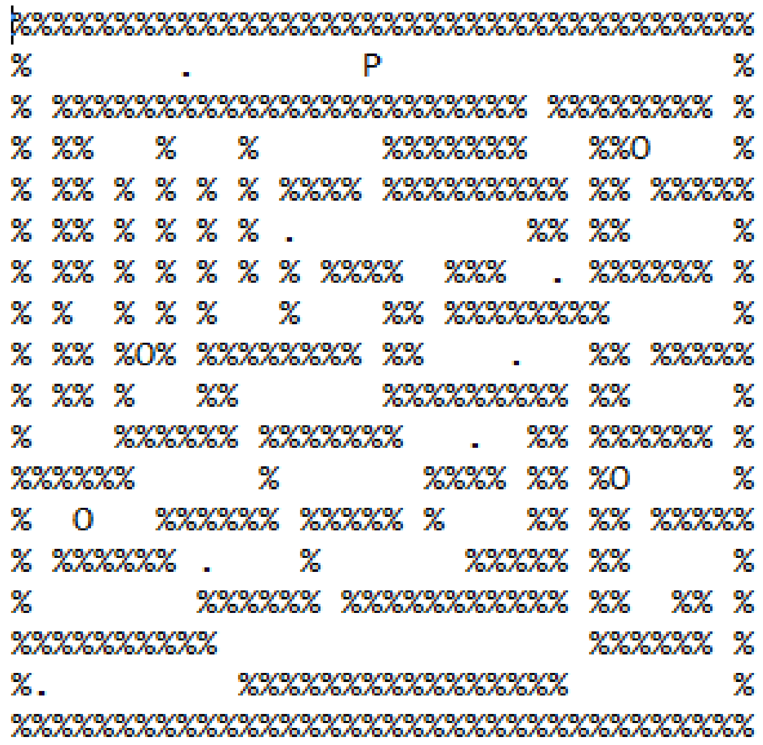
  - **__hash__():** Generates a unique hash for the game. Reduce the **overhead** when comparing visited game states → performance improvement!

```
RETURN hash((player position, frozen_set(pearls), frozen_set(gems), ghost_turns))
```

Class Game

- Essential methods:

  - **__str__():** Returns the game state as a string for console output.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       .        P                 %
% %%%%%%%%%%%%%%%%%%%%%%% %%%%%%%% %
% %%   %   %      %%%%%%%   %%O    %
% %% % % % % %%%% %%%%%%%%% %% %%%%%
% %% % % % % .           %% %%     %
% %% % % % % % %%%%  %%%  .  %%%%%% %
% %  % % %   %    %% %%%%%%%%      %
% %% %O% %%%%%%%% %%    .   %% %%%%%
% %% %   %%       %%%%%%%%% %%     %
%    %%%%%% %%%%%%%   .  %% %%%%%% %
%%%%%%      %       %%%% %% %O     %
%  O   %%%%%% %%%%% %    %% %% %%%%%
% %%%%%% .    %       %%%%% %%     %
%        %%%%%% %%%%%%%%%%% %%  %% %
%%%%%%%%%%                  %%%%%% %
%.         %%%%%%%%%%%%%%%%        %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 1. __str__()'s output

Class Pathfinder

- Defines an agent for finding the optimal path to collect all pearls in a game state. The agent uses A* search algorithm with a Minimum Spanning Tree (MST) heuristic.
- A MST is the most efficient way to connect all target nodes in a graph without forming any loops using the least total path cost.

Class Pathfinder

- The heuristic uses Prim algorithm to find the MST of a given game state:

1. Start at the player position (initial node).
2. Calculates a path cost of all reachable pearls and gems; then pick the shortest node.
3. Repeat step 2 at that node until all nodes are connected, forming the MST.

Class Pathfinder

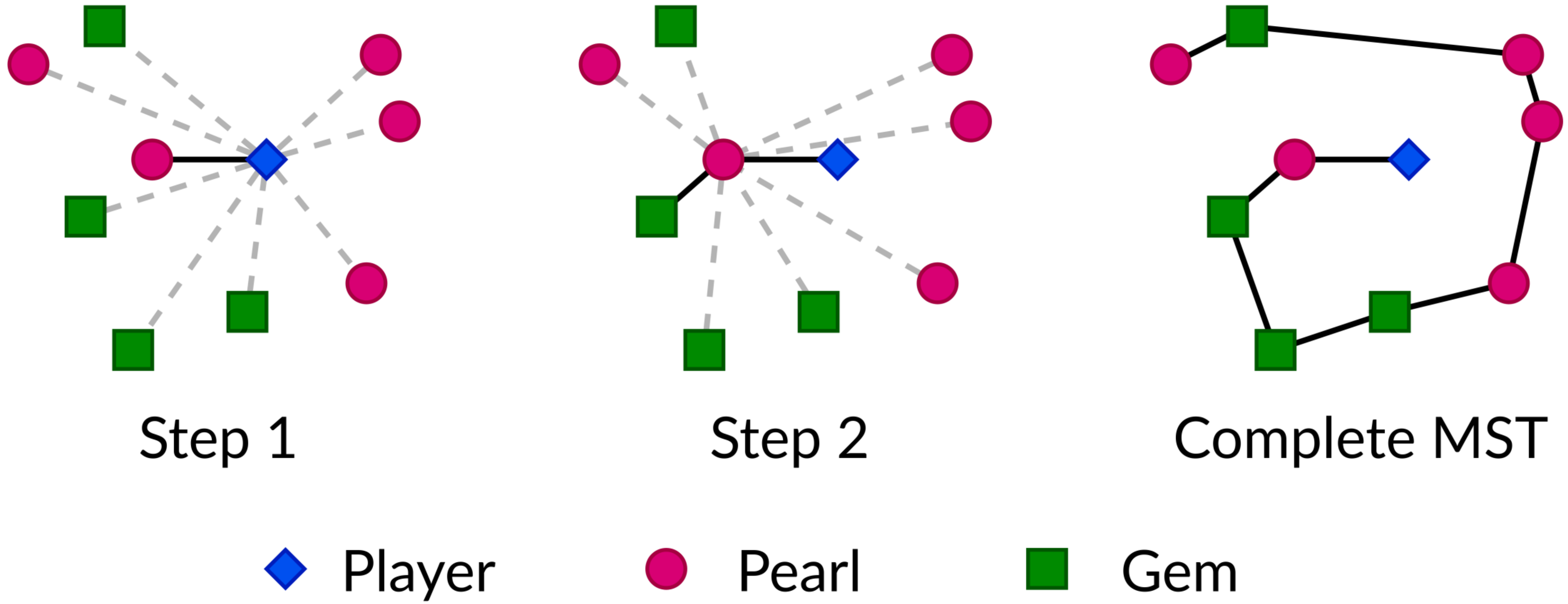


Figure 2. Prim Algorithm visualization

Class Pathfinder

- A* search with Prim MST Heuristic:

• Complete?	**YES**
• Optimal?	**YES**
• Time complexity?	*O(E*log(V))*
• Space complexity?	*O(V + E)*

(E is the number of edges, V is the number of vertices)

Class Pathfinder

- Essential methods:

  - **estimate():** Computes the MST cost to connect all remaining pearls, gems, and the player.

```
nodes = pearls + gems + [player]
IF node is empty:
    return 0
visited = set()
min_heap = MIN_HEAP([(0, nodes[0])])
```

Class Pathfinder

- Essential methods:

- **estimate():** (cont.)

```
WHILE heap has items AND NOT all nodes visited:
    cost, (x, y) = POP smallest item from heap
    IF (x, y) already IN visited:
        CONTINUE
    MARK (x, y) as visited
    ADD cost to total_cost
```

Class Pathfinder

- Essential methods:

  - **estimate():** (cont.)

```
FOR each (nx, ny) in nodes:
    IF (nx, ny) not in visited:
        distance = |nx - x| + |ny - y|
        PUSH (distance, (nx, ny)) to heap
RETURN total_cost
```

Class Pathfinder

- Essential methods:

  - **find():** Finds the shortest sequence of moves to collect all pearls.

```
frontier = [(self.estimate(self.src), 0, player, pearls, gems, ghost_turns, [])]
visited = SET()
WHILE frontier NOT EMPTY:
    f_cost, g_cost, player_pos, pearls_left, gems_left, path, ghost_turns =
            POP(frontier)
    game = Game(player_pos,….)
```

Class Pathfinder

- Essential methods:

  - **find():** (cont.)

```
IF hash(game) is NOT IN visited:
    visited.add()
IF pearls_left is EMPTY:
    RETURN path
```

Class Pathfinder

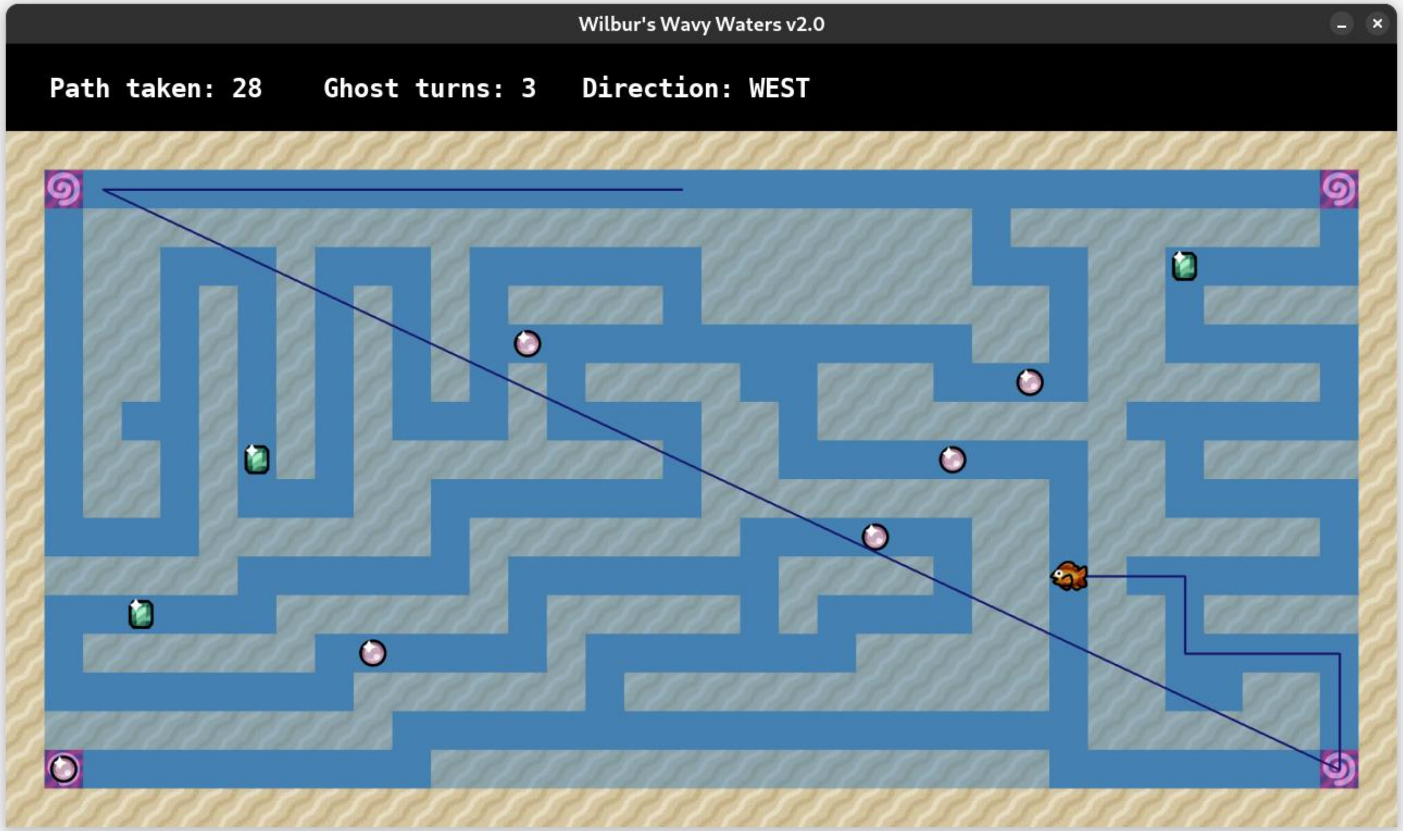
- Essential methods:

    - **find():** (cont.)

```
FOR direction, new_pos IN get_moves()
    new_game = game.move_to(new_pos)
    new_g_cost = g_cost + 1
    new_f_cost = new_g_cost + self.estimate(new_game)
    IF hash(new_game) not IN visited:
        heappush(frontier, (new_f_cost, new_g_cost, new_game.player,
  new_game.pearls, new_game.gems, new_game.ghost_turns, path +
  [direction]))
```

Class Rendering

– Uses pygame to visualise the game and the path Wilbur took!



```
Path taken: 28    Ghost turns: 3   Direction: WEST
```

Figure 3. pygame UI

II. IMPLEMENTATION:

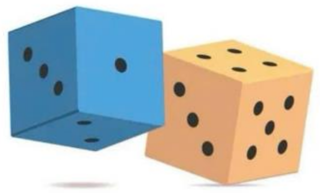
Task 3

Solution to 16-Queens with Genetic Algorithm

Essential libraries and modules

NUMPY

Numpy is used to create
and manipulate the chessboard as a 2D
array.



RANDOM

Random is used to
generate random integers for the initial
placement of queens on the chessboard.

Essential constants and parameters

BOARD_SIZE	Size of the chessboard and the number of queens to be placed on.
POPULATION_SIZE	The number of individuals (states) in the population for each generation.
MUTATION_RATE	The probability of mutation occurring in the population during each generation.
MAX_GENERATION	The maximum number of generations that the algorithm will run before stopping.

Class Chessboard

- This is a class to represent a chessboard in 2D array

- Attributes:

  • **board_size (int):** The size of the chessboard.

  • **board (numpy.ndarray):** A 2D array representing the chessboard.

- Every cell of the initial chessboard will be filled by a dot "."

- And the Queen will be visualize by a "Q".

- Index of column and row both start from 0.

Class Chessboard (cont.)

- Essential methods:

  - **place_queen(self, state):** This method is used to fills the board with dots and places queens ("Q") based on the provided state, which is a list of row indices for each column.

    Pseudocode (next slide)

Class Chessboard (cont.)

- Essential methods:

- **place_queen(self, state) - Pseudocode**

```
function place_queen(state) returns an 2D array
    fill every cell of the self.board with a dot "."
    iterate the state to get both index of col and row
        set the self.board[row, col] = "Q"
    end
```

Class Chessboard

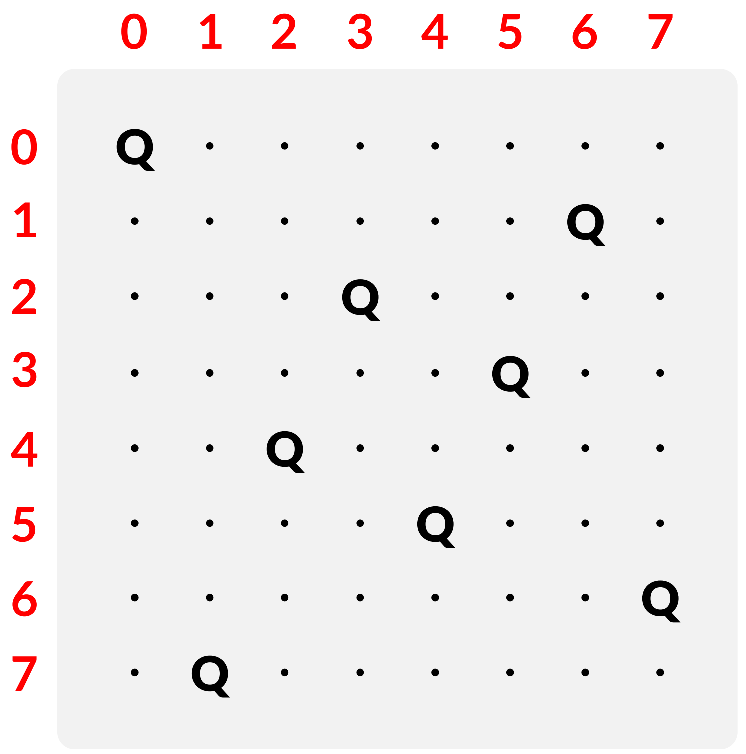

```
   0 1 2 3 4 5 6 7
0  Q . . . . . . .
1  . . . . . . Q .
2  . . . Q . . . .
3  . . . . . Q . .
4  . . Q . . . . .
5  . . . . Q . . .
6  . . . . . . . Q
7  . Q . . . . . .
```

Figure 4. 2D chessboard by numpy

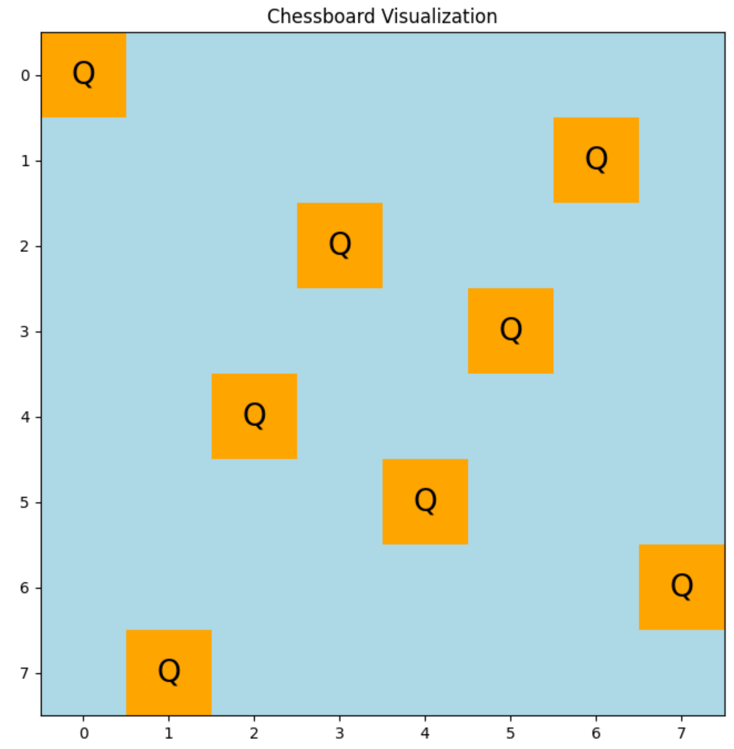


Figure 5. 2D chessboard by matplotlib

Class State

- This is a class to represent a state of chessboard in 1D array.
- Attributes:
    - **board_size (int):** The size of the chessboard.
    - **state (list):** A list of integers representing the row positions of queens.
    - **maximum_non_attacking_pairs (int):** The maximum number of non-attacking pairs of queens.

Class State (cont.)

- Attributes:

  - **fitness_score (int):** The fitness score of the state.
  - **selection_prob (float):** The selection probability of the state.
  - **length (int):** The length of the state

Class State (cont.)

- Essential methods:

  - **calculate_fitness(self)**: This method is used to calculate the fitness or number of non-attacking pair of each state.

  Pseudocode (next slide)

Class State (cont.)

- Essential methods:

- **calculate_fitness(self) - Pseudocode**

```
function calculate_fitness() returns an integer (numbers of non-attacking pair
in each state)
        initialize an array to store numbers of queen in each col
        initialize an array to store numbers of queen
        in main diagonal (from top-left to bottom-right)
        and anti diagonal (from bottom-left to top-right)
        initialize a variable to count numbers of attacking pair
        (attacking_pair, main_diagonal, anti_diagonal)
```

Class State (cont.)

- Essential methods:

- **calculate_fitness(self) - Pseudocode**

```
function calculate_fitness() returns an integer
        (initialize step)
        loop row <- in range of board_size
                col <- self.state[row]
                number_of_queens_col[col] ++
                main_diagonal[row - col + (board_size - 1) ] ++
                anti_diagonal[row - col + (board_size - 1) ] ++
```

Class State (cont.)

- Essential methods:

  - **calculate_fitness(self) - Pseudocode**

```
function calculate_fitness() returns an integer
        ( initialize and counting step)
        iterate count  in number_of_queens_col
                if count  > 1: attacking_pair += count * (count -1) // 2
        iterate count    in main_diangonal
                if count   > 1: attacking_pair += count * (count -1) // 2
        iterate number_of_queens  in anti_diangonal
                if count   > 1: attacking_pair += count * (count -1) // 2
```

Class State (cont.)

- Essential methods:

- **calculate_fitness(self) - Explaination**

  Take the 4 * 4 chessboard as an example. The number of diagonal from top-left to bottom-right is (2*board_size - 1) and the same to the diagoanl from bottom-left to top-right (Visualize in figure 3 and figure 4 in the next slide).

  After counting the appearance of queens in column, main diagonal and anti diagonal, apply the equation below to calculate the number of attacking pair:

  *number_of_attacking_pair = number_of queen * (number_of_queen - 1) // 2*

Class State (cont.)

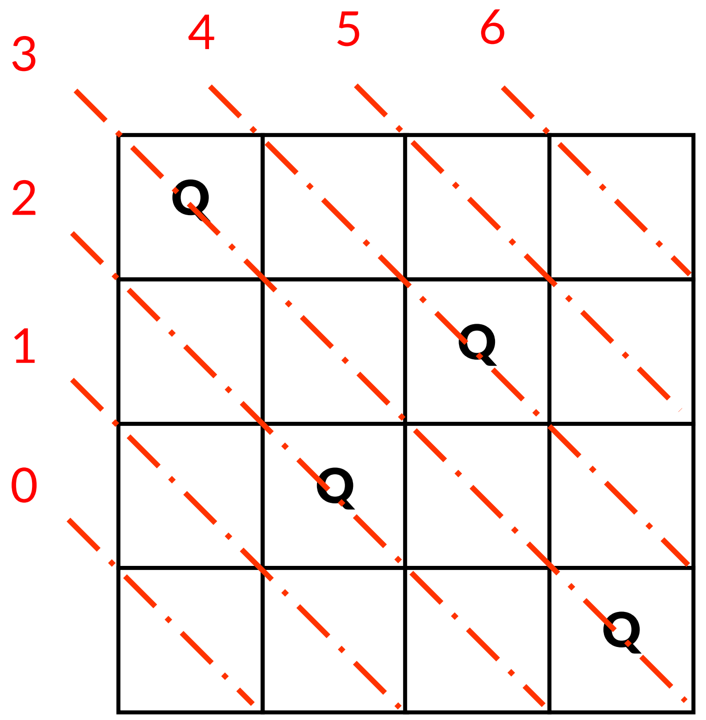


Figure 6. Main diagonal

-> [0, 0, 1, 2, 1, 0, 0]

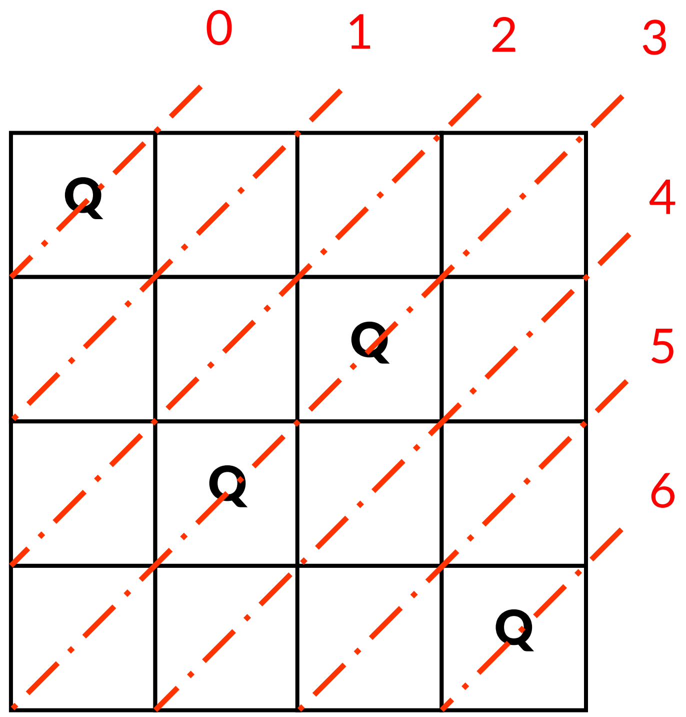


Figure 7. Anti diagonal

-> [1, 0, 0, 2, 0, 0, 1]

Class Population

- This is a class to represent the collection of state (generation).

- Attributes:

  - **board_size (int):** The size of the chessboard.
  - **population_size (int):** The size of the population (generation)
  - **mutation_rate (float):** Definition at the 3rd slide.
  - **max_fitness (int):** The max fitness score of each population.
  - **selection_prop (float):** The selection probability of each state.

Class Population (cont.)

- Essential methods:

  - **cross_over (self, parent1, parent2)**: This method is used to calculate the fitness or number of non-attacking pair of each state.

    Pseudocode (next slide)

Class Population (cont.)

- Essential methods:

  - **cross_over(self, parent1, parent2) - Pseudocode**

```
function cross_over(parent1, parent2) returns an two child state
    initialize start_point, end_point by random in range
    initialize allele1 <- parent1[start_point:end_point]
               allele2 <- parent2.copy()
               allele3 <- parent2[start_point:end_point]
               allele4 <- parent1.copy()
```

Class Population (cont.)

- Essential methods:

- **cross_over(self, parent1, parent2) - Pseudocode**

```
function cross_over(parent1, parent2) returns an two child state
    (initialize step)
    iterate number <- allele1
        iterate i in allele2
            if number == i  -> allele2.remove(i)
    iterate number <- allele3
        iterate i in allele4
            if number == i -> allele4.remove(i)
```

Class Population (cont.)

- Essential methods:

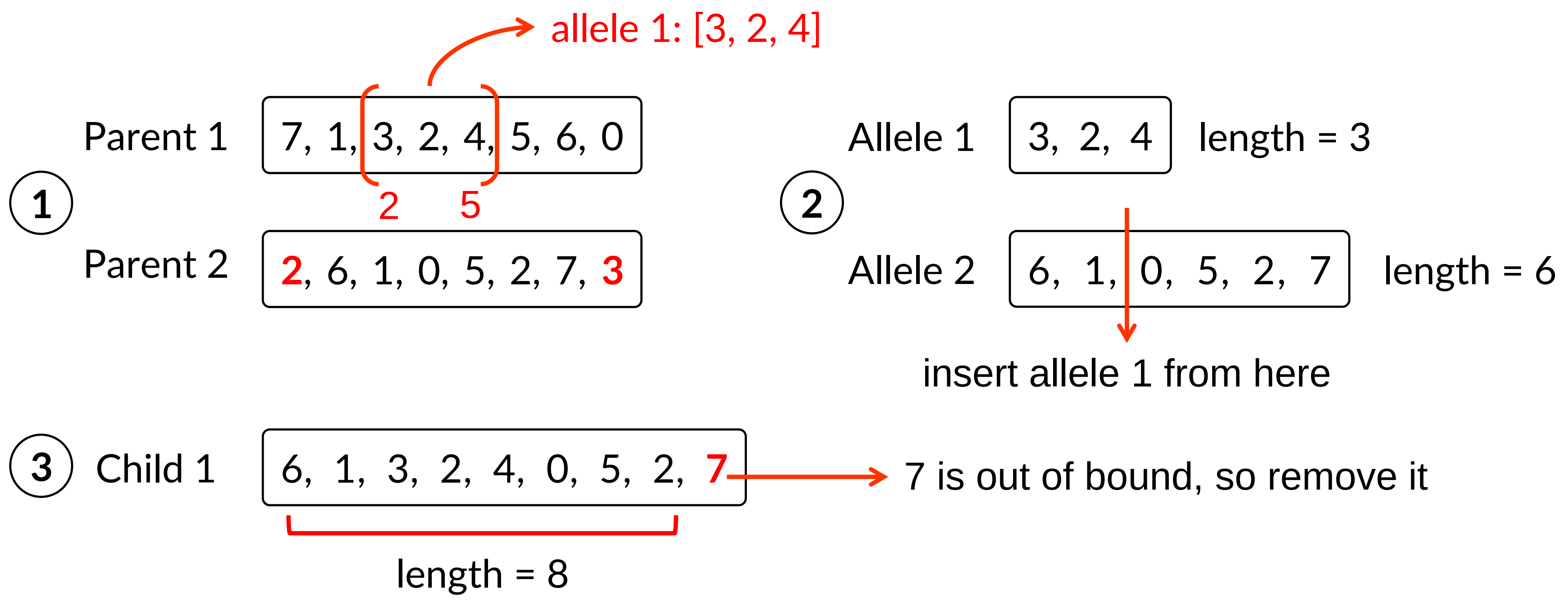
  - **cross_over(self, parent1, parent2) - Pseudocode**

```
function cross_over(parent1, parent2) returns an two child state
        (initialize and remove duplicate number in parent step)
        if len(allele1) + len(allele2) == 16
            child1 = allele2[:start_point] + allele1 + allele2[start_point]
        else
            child` = allele2[:start_point] + allele1 + allele2[start_point:end_point]
        same with child2
```

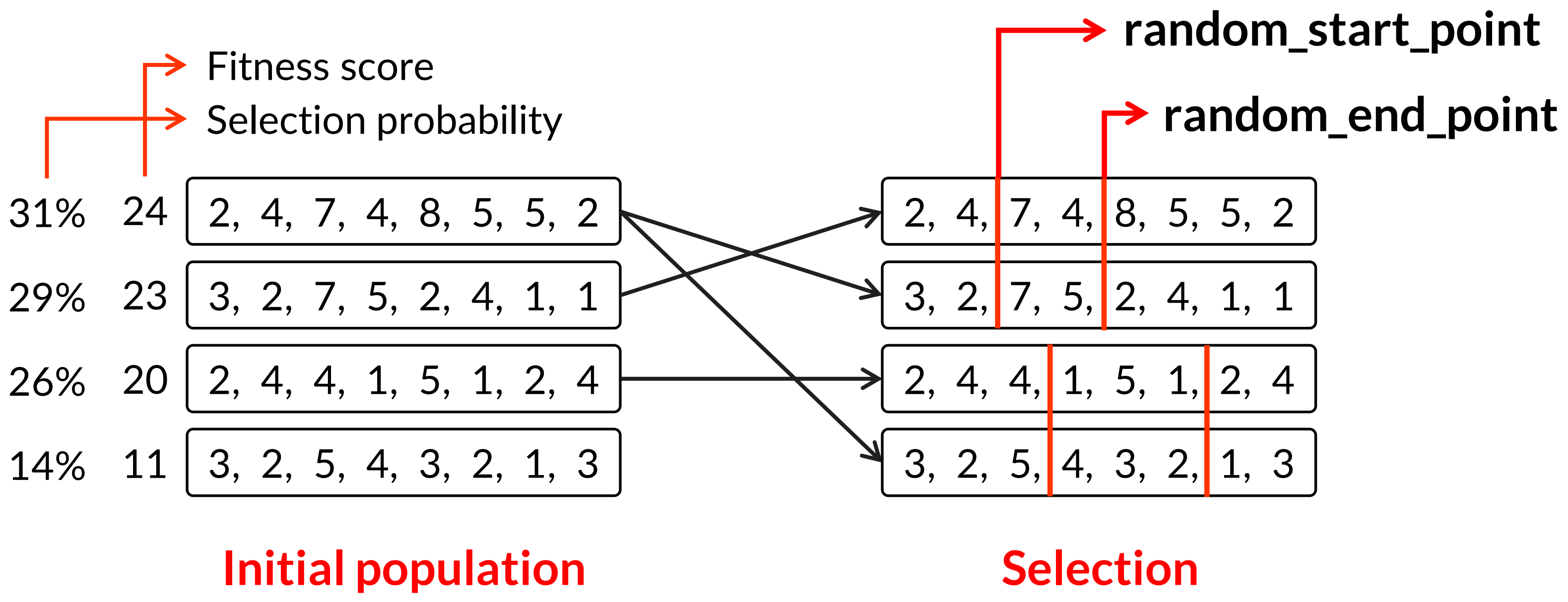
Class Population (cont.)

- Essential methods:

- **cross_over(self, parent1, parent2) - Explaination**



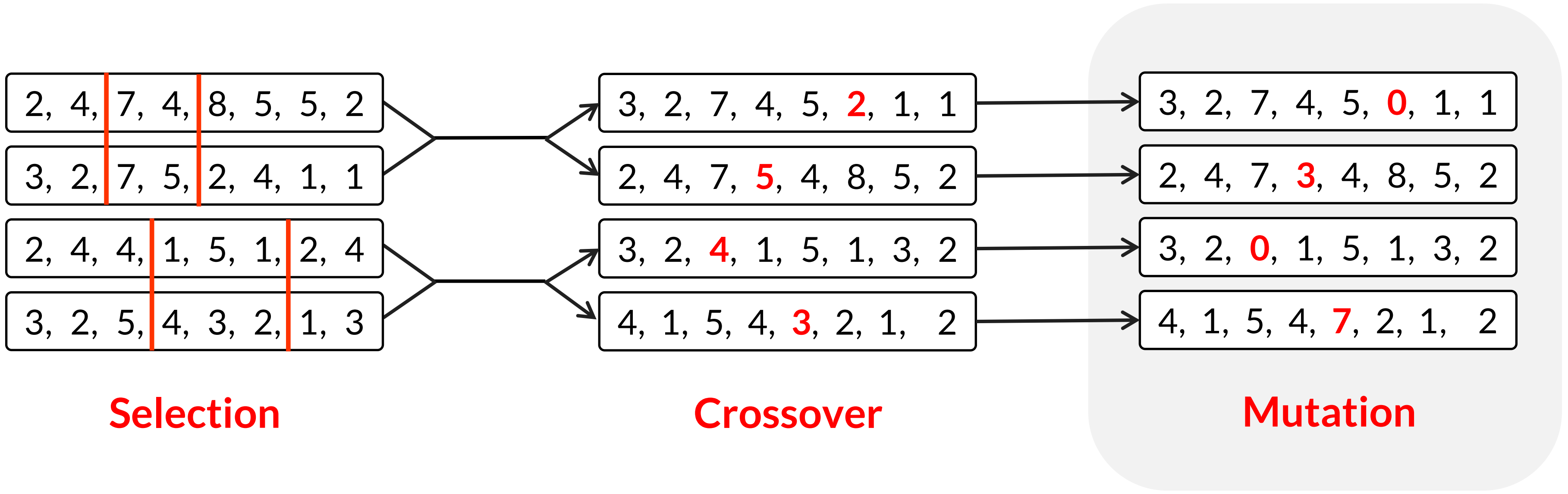
Overral



The state has the lowest selection probability will be eliminated

The state has the higest selection proabiltiy will replace the blank space

Overral (cont.)



NEW GENERTATION

If this generation does not contain the solution, return to **Selection** step