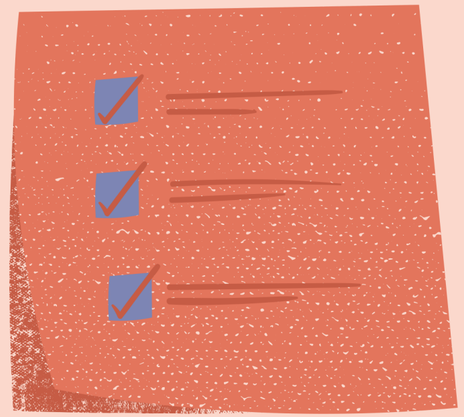
GROUP 6

Mid-term

Lecturers: Nguyen Thanh An

Table of contents



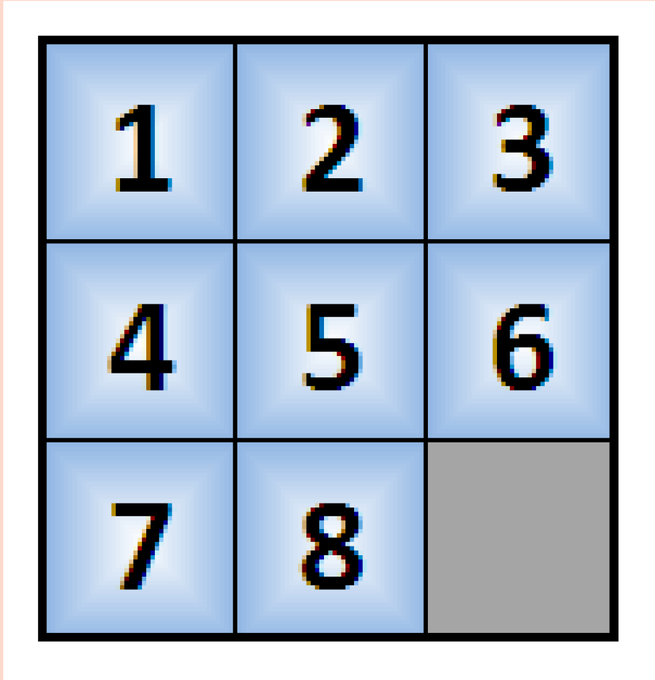
**Search Engine Optimization - Tối ưu hóa công cụ tìm kiếm là quá trình tăng chất lượng và lưu lượng truy cập website

I: Student list

ID	Full Name	Email	Assigned tasks	Percentage
520H0500	Trần Lê Minh Trí	520h0500@student.tdtu.edu.vn	Task 1	100%
522H0142	Nguyễn Thành Long	522h0142@student.tdtu.edu.vn	Task 1	100%
522H0148	Phạm Đặng Thanh Trung	522H0148@student.tdtu.edu.vn	Task 2	100%
520H0639	Trương Thái Gia Hưng	520H0639@student.tdtu.edu.vn	Presentation	100%

II:Approaches to solve tasks

8-puzzle with algorithm BFS



BFS

Class

Node

Graph_Search(BFS)

Random 1000

BFS

Node

Step to do

Initialize the Node class with attributes: state, parent, action, and cost.

Define the __str__ method to represent the state of the node as a string.

Define the __eq__ method to compare two nodes based on their states.

Define the get_blank_position method to find the position of the blank tile in the state.

Define the get_successors method to obtain the next possible states from the current state by moving the blank tile in the directions 'L' (left), 'R' (right), 'U' (up), and 'D' (down).

Define the get_successor method to retrieve the next state from an action and the current state.

BFS

Node

Step to do

Define the get_dest_pos method to calculate the destination position of the blank tile after moving in a certain direction.
Define the get_blank_pos method to find the position of the blank tile in a state.
Define the get_id method to retrieve the identifier of the node.
Define the get_node_str method to obtain the string representation of the node.
Define the draw method to visualize the node and its connections in a graph.
Define the is_goal_state method to check if the current state is a goal state.

BFS

Graph_Search

Step to do

Initialize the graph, frontier queue, explored set, solution path, total cost, and actions list.
Add the initial node and an empty path to the frontier queue.
Start a while loop until the frontier queue is empty.
Inside the loop, dequeue the current node and its corresponding path from the frontier.
Add the ID of the current node to the explored set.
Get the list of neighboring nodes for the current node using the get_successors() method.

BFS
Graph_Search

Step to do

For each neighboring node, check if it matches the goal node.
If a neighboring node is the goal node, construct the solution path by tracing back from the goal node to the initial node, while adding nodes and connecting edges in the graph. Also, store the actions taken in the actions list.
Display the length of the solution path below the goal node in the graph.
Return the graph, solution path, actions list, and total cost.
If the goal node is not found and the frontier queue becomes empty, return the graph, an empty solution path, an empty actions list, and the total cost.

BFS

Graph_Search

```
def graph_search(node, goal):
    dot = Digraph()
    frontier = deque()
    explored = set()
    frontier.append((node, []))  # Thêm solution_path vào frontier

    solution_path = []  # Mảng chứa solution path
    total_cost = 1  # Khởi tạo giá trị total_cost
    actions = []  # Danh sách hướng di chuyển
    while frontier:
        current_node, path = frontier.popleft()  # Lấy node và path từ frontier
        explored.add(current_node.get_id())

        successors = current_node.get_successors()
        for successor in successors:
            if successor.state == goal.state:
                current = successor
                while current is not None:
                    current_node_id = current.get_id()
                    dot.node(current_node_id, current.get_node_str()\
                             , shape='doublecircle')
                    if current.parent is not None:
                        dot.edge(current.parent.get_id() \
                                 , current_node_id, label=current.action)
                        actions.append(current.action)
```

BFS

Graph_Search

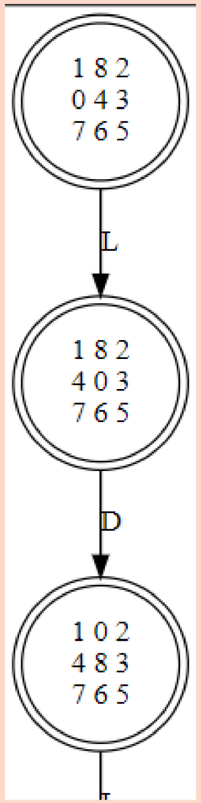
```
            solution_path.append(current)
            # Thêm state vào solution path
            current = current.parent

        # Hiển thị độ dài của solution path phía dưới goal state
        path_cost = len(solution_path) - 1
        dot.node(f'Path_Cost_{path_cost}' \
                 , label=f'Path Cost: {path_cost}' \
                 , shape='plaintext')
        dot.edge(successor.get_id()\
                 , f'Path_Cost_{path_cost}', label='')

        return dot, solution_path, actions, total_cost
        # Trừ đi trạng thái ban đầu từ solution path

    if successor.get_id() not in explored and \
     (successor, path + [successor]) not in frontier:
      # Kiểm tra cả path khi thêm vào frontier
      frontier.append((successor, path + [successor]))
     # Thêm node và path vào frontier
      total_cost += 1
    # Tăng giá trị total_cost lên 1 mỗi khi thêm một cạnh vào frontier

return dot, solution_path, actions, total_cost
```

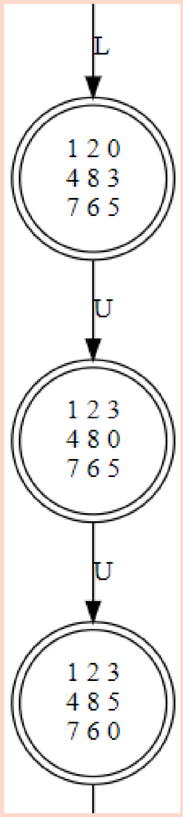
BFS

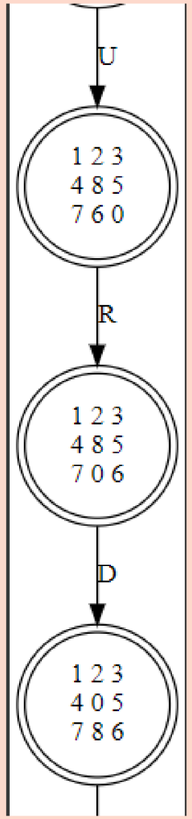
Result

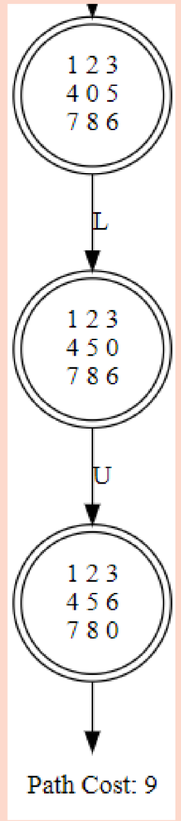
Left -> Right

```
Actions: ['U', 'L', 'D', 'R', 'U', 'U', 'L', 'D', 'L']
Total Cost: 385
```









BFS

Random 1000

In the case of a puzzle like the 8-puzzle, there are certain configurations of the puzzle that are unsolvable. These configurations result in what is known as an "unsolvable state."

BFS

Random 1000

Example

initial state

2 1 3
4 5 6
7 8 0

goal state

1 2 3
4 5 6
7 8 0

BFS
Random 1000

Check random state

```
def count_inversions(state):
    inversions = 0
    n = len(state)

    flattened_state = [num for row in state for num in row if num != 0]

    for i in range(len(flattened_state)):
        for j in range(i + 1, len(flattened_state)):
            if flattened_state[i] > flattened_state[j]:
                inversions += 1

    return inversions

if not is_valid_state(puzzle_state):
    return False

inversions = count_inversions(puzzle_state)

if inversions % 2 == 0:
    return True
else:
    return False
```

BFS
Random 1000

Random state

```
def randomize_puzzle():
    puzzle = Node([[1, 8, 2], [0, 4, 3], [7, 6, 5]])
    moves = [(0, 1, 'L'), (0, -1, 'R'), (1, 0, 'U'), (-1, 0, 'D')]

    for _ in range(10):
        move = random.choice(moves)
        blank_i, blank_j = puzzle.get_blank_position()
        new_i, new_j = blank_i + move[0], blank_j + move[1]

        if 0 <= new_i < len(puzzle.state) and 0 <= new_j < len(puzzle.state[0]):
            puzzle.state[blank_i][blank_j] \
            , puzzle.state[new_i][new_j] = puzzle.state[new_i][new_j] \
            , puzzle.state[blank_i][blank_j]

    return puzzle.state
```

BFS

Random 1000

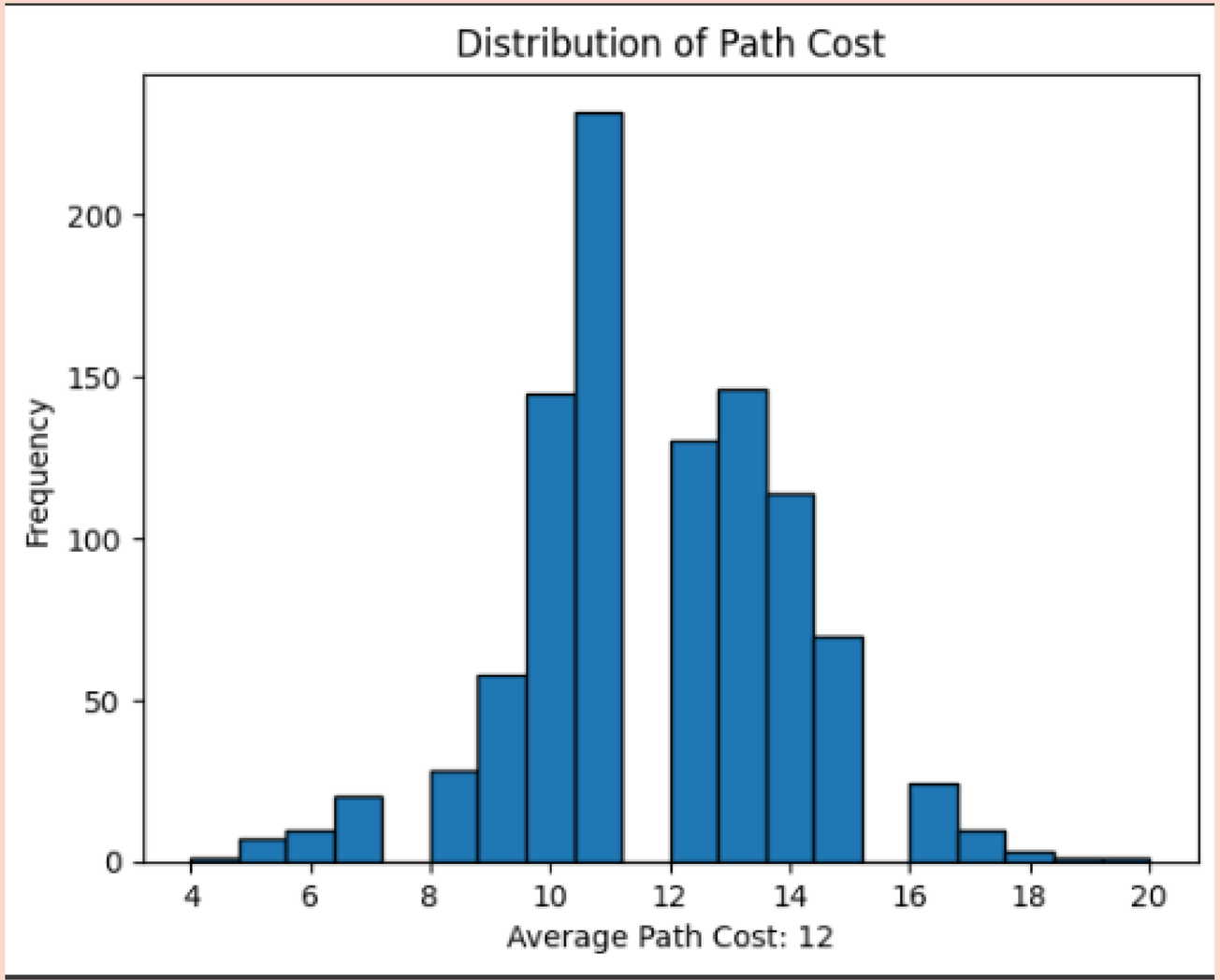
Avarage patth cost

```
# Generate 1000 random puzzles
random_puzzles = [randomize_puzzle() for _ in range(1000)]
goal1 = Node([[1, 2, 3],
         [4, 5, 6],
         [7, 8, 0]])

# Create a list of goal states
goal_states = [goal1, goal2]
# Calculate total cost for each puzzle
total_costs = []
total_cost= 0

for puzzle in random_puzzles:
    start_node = Node(puzzle)
    dot, solution_path, action, total = graph_search(start_node, goal1)
    total_cost = len(solution_path)
    total_costs.append(total_cost)


average_path_cost = sum(total_costs) / len(total_costs)
plt.hist(total_costs, bins=20, edgecolor='black')
plt.xlabel(f'Average Path Cost: {average_path_cost:.0f}')
plt.ylabel('Frequency')
plt.title('Distribution of Path Cost')
plt.show()
```

BFS

Random 1000

Result



BFS

Practical Examples

Suppose you are in a city with a complex road network and you want to get from point A to point B. You want to find the shortest route in the traffic system to save time and energy.

BFS

Practical Example

Advantages

1. Finding the shortest path: BFS ensures finding the shortest path between two locations in the city, if a path exists.
2. Easy implementation: The BFS algorithm has a simple and straightforward structure, making it easy to implement for pathfinding in a city.

Disadvantages

1. Memory consumption: BFS requires storing all explored locations in a queue. This can be memory-intensive, especially in large cities or with a large number of locations.
2. Long runtime: In large cities, executing the BFS algorithm can take a significant amount of time to find the path, particularly when there are numerous locations and complex road networks.

A star

Core Data Structure

PuzzleState Class:
    Properties:
        goal_state = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])

    Methods:
        __init__(self, state, parent=None, action=None, depth=0):
            Initialize state, parent, action, depth, heuristic_cost, and total_cost attributes with provided values.
            Calculate the heuristic_cost using the calculate_manhattan_distance() method.

***Solution:**
In the PuzzleState class, we initialize a state of the puzzle game with basic properties such as current state, parent state, action, depth, cost of heuristics based on Manhattan distance, and total cost of the state.

Importance of the Manhattan distance for heuristic cost calculation.

```
calculate_manhattan_distance():
    Initialize total_distance to 0
    For each number from 1 to 8:
        Find the position of the number in the current state
        Find the position of the number in the goal state
        Calculate the Manhattan distance between the positions
        Add the Manhattan distance to the total_distance
    Return the total_distance as the heuristic cost
```

***Solution:**

This function calculates the Manhattan distance between the current state and the target state by going through each number in the range 1 to 8, then calculates the Manhattan distance for each number and adds it to the total distance and returns the value. total distance value.

How A* Search Finds Solutions

```
Function a_star_search(initial_state):
    open_set = PriorityQueue()
    counter = itertools.count()
    start_state = PuzzleState(initial_state)
    open_set.put((start_state.total_cost, next(counter), start_state))
    visited = set()

    while open_set is not empty:
        current_cost, _, current_state = open_set.get()

        If current_state is the goal state:
            Return current_state

        Add the current_state's state to the visited set

        For each move in current_state's possible moves:
            Generate the successor state by making the move
            If the successor's state is not in the visited set:
                Add the successor to the open_set with its total_cost as the priority

    Return None
```

***Solution:**

The above code implements the A* search algorithm in solving the problem. It uses a priority queue to cycle through the problem states in order of priority. The algorithm will browse through neighboring states, calculate costs and evaluate to find the optimal path from the initial state to the goal state.

Output: list of actions (Left, Right, Up, Down); total cost

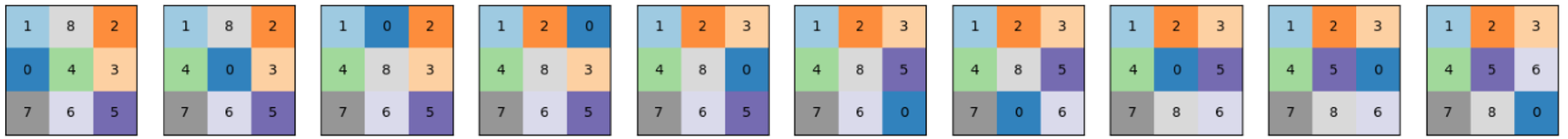
Function main():
    Set initial_state to [[1, 8, 2], [0, 4, 3], [7, 6, 5]]

    Obtain the solution by invoking a_star_search(initial_state)

    If a solution is found:
        Extract the actions and total_cost by calling the print_solution() function on the solution
        Display "Solution found with the following moves:" followed by actions
        Display "Total cost (depth + heuristic):" followed by total_cost
        Invoke the plot_solution() function with the solution
    Else:
        Display "No solution found."

Call main() to start the program

```
Solution found with the following moves: ['Right', 'Up', 'Right', 'Down', 'Down', 'Left', 'Up', 'Right', 'Down']
Total cost (depth + heuristic): 9
```



Generate randomly 1000 initial states

shuffle_state: Generates a random puzzle state.

```
Function shuffle_state(state, moves=100):
    current_state = create PuzzleState object with state
    For _ in range(moves):
        randomly select a move from the possible moves of current_state
        Update current_state by generating its successor based on the selected move
    Return the state of current_state
```

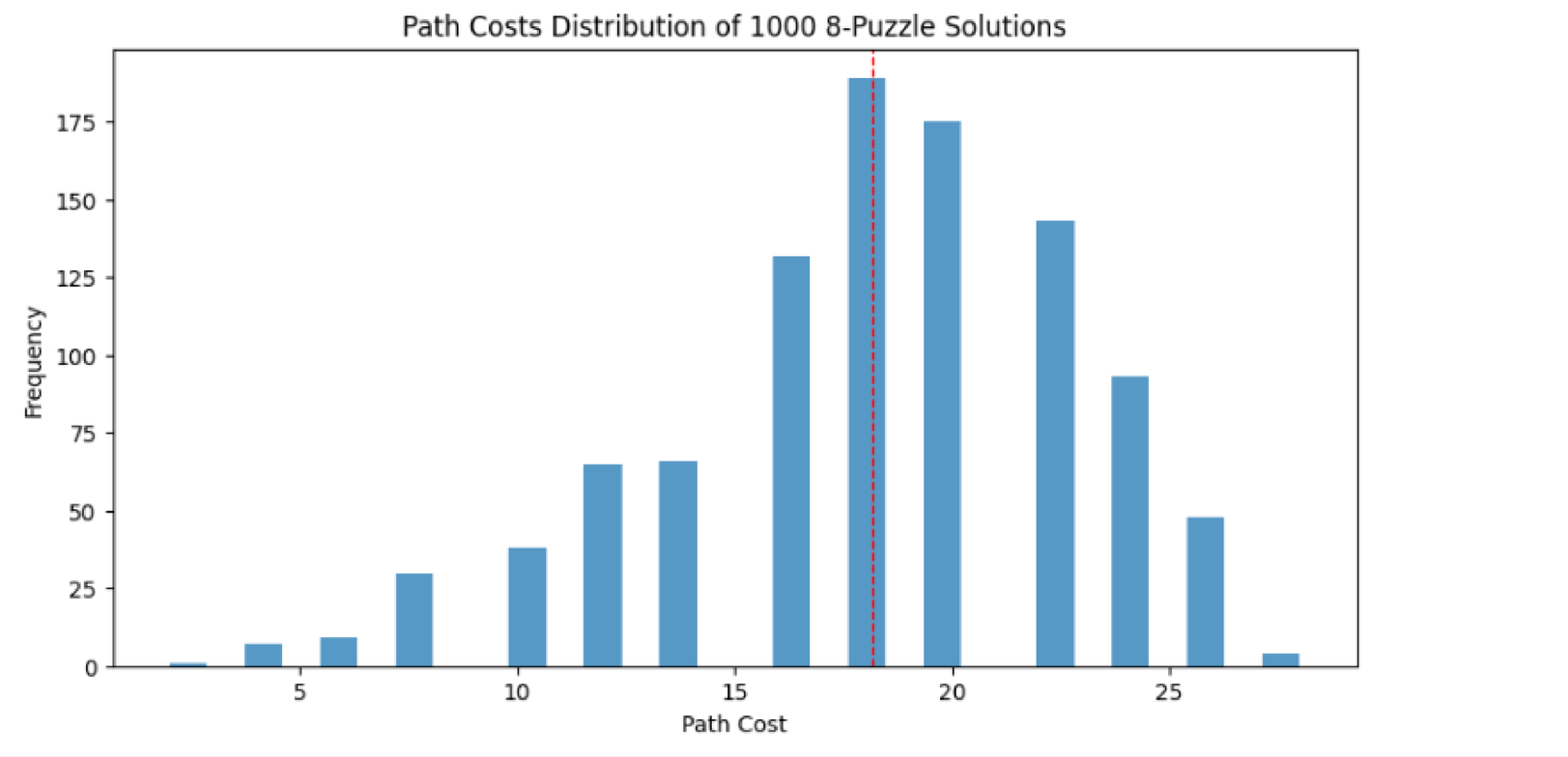
generate_random_states: Produces a list of random states.

```
Function generate_random_states(n=1000):
    random_states = array of n elements
    For each element in random_states:
        Assign the shuffled state of the PuzzleState.goal_state with 100 random moves
    Return random_states
```

solve_and_analyze_states: Solves the states and analyzes the results.

```
Function solve_and_analyze_states(states):
    Initialize an empty list path_costs
    For each state in states:
        Execute A* search to find a solution for the current state
        If a solution is found:
            Retrieve the total_cost from printing the solution
            Add the total_cost to the path_costs list
    Calculate the average_cost as the sum of path_costs divided by the number of path_costs
    Return the average_cost and the list of path_costs

random_states = generate_random_states(1000)
average_cost, path_costs = solve_and_analyze_states(random_states)
```

Visualize results



Practical Example

Imagine you have a mobile robot tasked with navigating a cluttered warehouse to pick up items and deliver them to different locations. The robot needs to find the shortest path to each destination while avoiding obstacles and optimizing its movement.

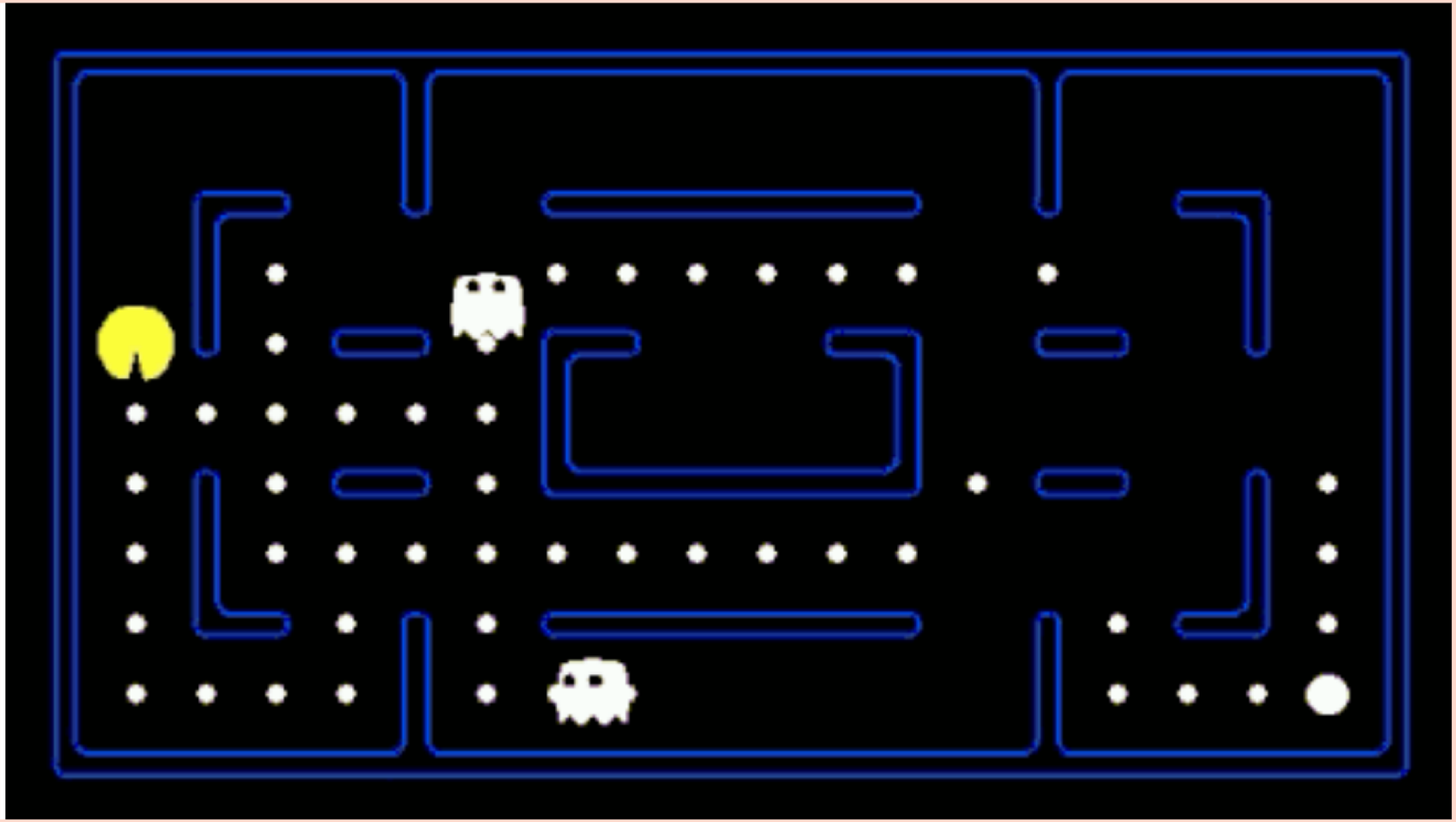
Advantages

1. Efficiency: A* utilizes heuristics (such as Euclidean distance or Manhattan distance) to estimate the remaining cost to reach the goal, which helps prioritize paths more likely to lead to the shortest path. This can significantly reduce the search space and improve the efficiency of the robot's navigation.
2. Optimality: Under certain conditions, A* guarantees finding the shortest path between the robot's current location and the target destination, ensuring efficient item retrieval and delivery in the warehouse.
3. Flexibility: The A* algorithm allows for customization by incorporating different heuristics and cost functions based on specific requirements. This flexibility enables fine-tuning the robot's navigation behavior to suit the warehouse's characteristics and constraints.

Disadvantages

1. Memory and computational requirements: Depending on the size and complexity of the warehouse, the A* algorithm may require substantial memory and computational resources. Storing and searching large graphs representing the warehouse layout can become challenging, especially if the map is dynamically changing.
2. Heuristic accuracy: The quality of the heuristic function used in A* can greatly impact the algorithm's performance. Inaccurate or poorly chosen heuristics may lead to suboptimal paths or inefficient navigation for the robot.
3. Dynamic environments: If the warehouse environment is dynamic, with moving obstacles or changing paths, the A* algorithm may need to be adapted to handle real-time updates and efficiently replan paths to avoid collisions or obstacles.

Pacman with the A* Alorithm and UCS Algorithm



Source: https://blog.sciencemuseum.org.uk/pac-man-turns-40/

Pacman

Core Data Structure

Maze Class: Reads a maze from a given file, identifying the starting point ('P'), food locations ('.'), and walls ('%'). It provides methods to get the initial state, targets (food locations), and possible actions from any given state.

Priority Queue (PQueue) Class: Implements a priority queue to manage states in UCS and A* algorithms based on their cumulative cost (for UCS) or estimated total cost (for A*).

Priority Queue (PQueue) Class: Implements a priority queue to manage states in UCS and A* algorithms based on their cumulative cost (for UCS) or estimated total cost (for A*).

Pacman

UCS

Goal: Find the least costly path.

Pseudo-code:
Initialize the priority queue with the initial state.
While the queue is not empty, do the following:

- Dequeue the lowest-cost state.
- If it's a goal state, return the path to it.
- Otherwise, expand the state and enqueue all successors with their cumulative costs.

Return failure if no path is found.

Pacman

A-Star

Goal: Find the least costly path using a heuristic to estimate costs.

Pseudo-code:
Initialize the priority queue with the initial state.
Set the cost from start to the initial state to zero.
While the queue is not empty, do the following:

- Dequeue the state with the lowest estimated total cost.
- If it's a goal state, return the path to it.
- Otherwise, expand the state and enqueue all successors with their cost from start plus the heuristic estimate to the goal.

Return failure if no path is found.

Practical Example

GPS routing system or mobile navigation applications.

Search for locations

Positioning and directions

Live traffic information

Update directions directly

Advantages

A*

- Fastest route finding: The A* algorithm can be used to find the optimal path between two points on a map. For instance, when you input location A and location B into a navigation app, the A* algorithm calculates and suggests the most efficient route based on current traffic information and distances between points.
- Path customization: A* can be enhanced to consider various factors such as road closures, waypoints, or points of interest along the route. This helps create a customized route to cater to individual user preferences.

UCS

- Lowest-cost route finding: UCS is used to find the path with the lowest cost between points on a map. When applied to a GPS routing system, the UCS algorithm can suggest a route based on distance and estimated travel time between locations.
- Accuracy: UCS ensures accurate path finding from one location to another based on specific cost information.

Disadvantages

A*

- Computational requirements: The A* algorithm may require significant computation and computational resources to find the optimal path, especially on large or complex maps. This can impact system performance and response time.
- Accuracy of positioning information: The performance of the A* algorithm relies on the accuracy of positioning information and map data. If the information is inaccurate, the algorithm may propose suboptimal or unrealistic routes.

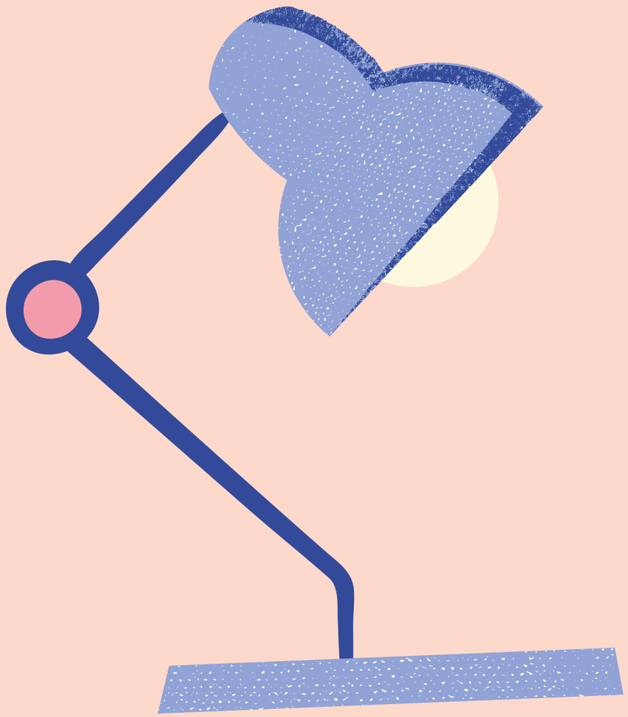
UCS

- No time prioritization: UCS does not consider travel time or current traffic conditions, which may result in suboptimal time-wise routes.
- External factors not considered: UCS does not evaluate external factors such as road closures, waypoints, or other amenities along the route. This may limit customization and personal utility in route selection.

VI:Table of complete percentages

Table of complete percentages

Missions	Percentages
Task 1	Completed
Task 2	Completed



References

https://ai.stackexchange.com/questions/9182/how-do-i-show-that-uniform-cost-search-is-a-special-case-of-a

https://www.youtube.com/watch?v=_CrEYrcImv0

https://www.cs.cmu.edu/~maxim/files/mha_ijrr15.pdf

THANKYOU