Midterm presentation

Subject: Introduction to Artificial Intelligence

Group: 24

Instructor: Msc **Nguyễn Thành An**

Introduction to Graph Traversal Algorithms

– **Best – Frist Search** (BFS): frontier is ***FIFO queue***

– **Uniform Cost Search** (UCS): frontier is ***Priority Queue*** order by path cost, g(n)

– **A Star** (A*): frontier is ***Priority Queue*** order by path cost + heuristic function, g(n) + h(n)

Introduction to 8-Puzzel Solving Problem

8-Puzzle solving algorithms are vital in AI and problem-solving, used to find optimal solutions in gaming, robotics, and decision-making, including informed searches and uninformed searches like A star, BFS, UCS.

Overview of the 8 puzzle problem

Initial State

The 8 puzzle problem starts with a configuration of 8 numbered tiles on a board, with one empty space.

Goal Test

The goal state is the configuration where the tiles are arranged in ascending order, with the empty space in the bottom right corner or the top left corner.

Actions

Movements of the blank space with one of the actions Left, Up, Right, Down depend on the location of the blank space.

Transition Model

Return a resulting state given a state and an action.

Path Cost

Each action costs 1 step.

Summary of how to solve the requirements of the 8-puzzle

- Class Problem
  - __init__
  - __lt__
  - __str__
  - get_successors
  - get_successor
  - get_dest_pos
  - get_blank_pos
  - get_id
  - get_node_str
  - get_action
  - draw
  - goal_test
  - manhattan_distance

- Class SearchStrategy
  - search_BFS
  - search_AStar
  - visualize
  - get_actions
- Main
  - run_algorithm
  - generate
  - generate_initial_state
  - draw_bar_chart
  - Some other support functi

Summary of how to solve the requirements of the 8-puzzle

- Class Problem → **Problem store node**
  - __init__ → Node(state, id, parent, action, cost)
  - __lt__ → Comparison node costs for Priority Queue
  - __str__
  - get_successors
  - get_successor
  - get_dest_pos
  - get_blank_pos
  - get_id
  - get_node_str
  - get_action
  - draw → Draws the node and connection to its parent
  - goal_test → Check goal state
  - manhattan_distance → Heuristic function for A*

Other helper functions

Summary of how to solve the requirements of the 8-puzzle

- Class SearchStrategy → **Search Agent**
  - search_BFS → Algorithm BFS
  - search_AStar → Algorithm A*
  - visualize → Visual list actions of Pacman
  - get_actions → Get list actions of Pacman

Summary of how to solve the requirements of the 8-puzzle

- Main → **Main**
  - run_algorithm → Program execution function
  - generate → Generates a random state
  - generate_initial_state → Generates a random state based on moving from the goal state and randomly moving the blank tile
  - draw_bar_chart → Draws a bar chart of time and cost for the two algorithms using *matplotlib* library.
  - Some other support functions

BFS pseudocode for 8 - Puzzle

```
Function BFS(initial_node) returns current_node, actions
        frontier ← a FIFO queue with node as the only element
        explored ← an empty set
        frontier ← put initial_node
        Loop do
                if EMPTY?(frontier) then returns Failure
                current_node ← GET(frontier)
                add str(current_node) to explored
                if GOAL_TEST(current_node) then
                    return current_node, actions
                for each successor in current_node.get_successors() do
                    if successor not in explored and frontier then
                        frontier ← put(successor)
```

A* pseudocode for 8 - Puzzle

Function manhattan_distance(state) **return** distance

distance = 0

for each cell in state:

If cell not blank:

i = cell.row

j = cell.column

goal_i, goal_j ← divmod(value - 1, 3) or divmod(value, 3). Choose a smaller distance.

distance += abs(i - goal_i) + abs(j - goal_j)

A* pseudocode for 8 - Puzzle

```
Function Astar(initial_node) returns current_node, actions
    frontier ←  a priority queue with node and order by fn
    explored ← an empty set
    frontier ← put (0, initial_state)
    Loop do
        if EMPTY?(frontier) then returns Failure
        current_node ← GET(frontier)
        add current_node to explored
        if GOAL_TEST(current_node) then return current_node, actions
        for each successor in current_node.get_successors() do
            if successor not in explored and not in frontier:
                successor.cost ← currnet_node.cost + step_cost
                fn←  successor.cost + heuristic(successor.state)
                frontier ← put(fn, successor)
```

Propose a heuristic functions

The Manhattan distance heuristic [2]:

The Manhattan distance heuristic returns the sum of the Manhattan distances of each cell in the 8 puzzle's 8-tile configuration from its correct position in the goal state.

The admissibility property of the Manhattan heuristic $\Leftrightarrow \boldsymbol{h(n) \leq h^*(n)}$ where

- $\boldsymbol{h(n)}$ is the Manhattan distance of state $\boldsymbol{n}$.
- $\boldsymbol{h^*(n)}$ is the actual cost to reach the goal state.

This means:

- If $\boldsymbol{h(n) = 1}$, then $\boldsymbol{h^*(n)}$ must be at least 1 or greater.
- If $\boldsymbol{h(n) = 2}$, then $\boldsymbol{h^*(n)}$ must be at least 2 or greater.
- If $\boldsymbol{h(n) = 3}$, then $\boldsymbol{h^*(n)}$ must be at least 3 or greater.

Thus, for **every state n**, **we have** $\boldsymbol{h(n) \leq h^*(n)}$**,** which satisfies **the admissibility property for the Manhattan distance heuristic.**

Propose a heuristic functions

The Manhattan distance heuristic returns the sum of the Manhattan distances of each cell in the 8 puzzle's 8-tile configuration from its correct position in the goal state.

The consistency property of the Manhattan heuristic states that it is consistent $\Leftrightarrow \boldsymbol{h(n) \leq c(n, a, n') + h(n')}$, where

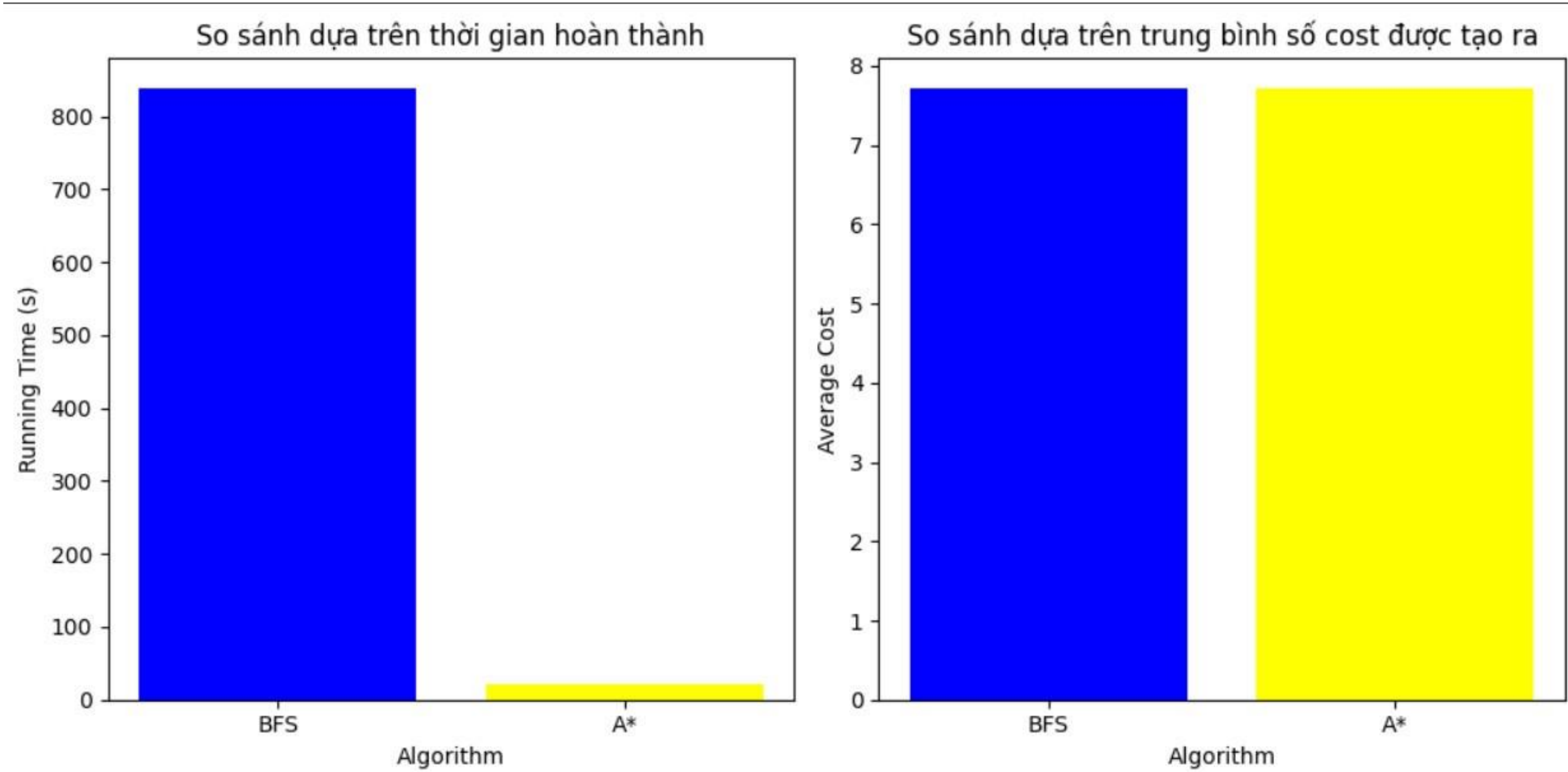
- $\boldsymbol{h(n)}$ is the Manhattan distance of state $\boldsymbol{n}$
- $\boldsymbol{c(n, a, n')}$ is the cost of applying action $\boldsymbol{a}$ from state $\boldsymbol{n}$ to state $\boldsymbol{n'}$
- $\boldsymbol{h(n')}$ represents the Manhattan distance cost of state $\boldsymbol{n'}$

This means:

- If $\boldsymbol{h(n) = 1}$, then $\boldsymbol{h(n')}$ must be at least 0 or greater.
- If $\boldsymbol{h(n) = 2}$, then $\boldsymbol{h(n')}$ must be at least 1 or greater.
- If $\boldsymbol{h(n) = 3}$, then $\boldsymbol{h(n')}$ must be at least 2 or greater.

Therefore, for **every state** $\boldsymbol{n}$, we have $\boldsymbol{h(n) \leq c(n, a, n') + h(n')}$**,** which satisfies **the consistency property for the Manhattan distance heuristic.**

Comparison of A* and BFS Algorithms for Solving the 8 Puzzle Problem by using Chart




```
Random 1000 initial state:
Trung bình số cost của thuật toán BFS: 7.718
Trung bình số cost của thuật toán A* với hàm manhattan heuritstic: 7.718
Thời gian thực thi 1000 trạng thái ngẫu nhiên của BFS: 838.2566857337952
Thời gian thực thi 1000 trạng thái ngẫu nhiên của A*: 20.391765594482422
```

Comparison of A* and BFS Algorithms for Solving the 8 Puzzle Problem

A* Algorithm	BFS Algorithm
is an informed search algorithm	is an uninformed search algorithm
uses heuristics to efficiently find the optimal solution	explores all possible states
ability to consider the likely shortest path makes it more efficient for **large state spaces.**	has the advantage of finding the **shallowest solution**, but it may not be optimal in terms of the number of moves required.
requires more memory and computational resources	requires less memory-intensive
Conclusion: BFS can be slower than A* (due to its exhaustive exploration of states) A* is more optimal than BFS (provided the Heuristic function satisfies admissibility and consistency properties).	

ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Introduction to the Pacman game

Overview of the Pacman problem

Initial State

The map consists of walls, 1 Pacman, and food dots located at various positions on the map.

Goal Test

Pacman eats all food dots and visits all 4 corners of the walls in any order.

Actions

Movements of Pacman with one of the actions North, East, West, South, and Stop.

Transition Model

The state of the map will update, and the position of Pacman will be updated when performing an action.

Path Cost

Each action taken by Pacman incurs a cost of 1.

Summary of how to solve the requirements of the Pacman

- MapGame.py
  - __init__
  - load_map
  - __str__
  - get_successors
  - add_dots_to_corners
  - update_map
  - get_blank_pos
  - get_start_state
  - is_dot
  - find_nearest_goal_state
  - is_goal_state
  - manhattan_distance
- SearchStrategy.py
  - Search_AStar
  - Search_UCS
  - visualize_pacman_movement
  - clear_screen
  - apply_action
  - print_map
- Test.py
  - run_algorithm

Summary of how to solve the requirements of the Pacman

- MapGame.py → **Map in game**
  - __init__ → MapGame(map_file)
  - load_map → Reads the map and stores in a 2D array
  - get_start_state → Retrieves Pacman's position.
  - get_successors → Retrieves positions Pacman can move.
  - add_dots_to_corners → Finds positions of dots in the four corners of the map.
  - update_map → Removes dots from the map after Pacman eats them.
  - get_blank_pos
  - __str__
    - Other support functions
  - is_dot → Check current state is dot?.
  - find_nearest_goal_state → Find nearest dot.
  - is_goal_state → Map don’t have any dot.
  - manhattan_distance → Heuristic of A*

Summary of how to solve the requirements of the Pacman

SearchStrategy.py → **The parent class of two algorithm classes UCS and A*.**

- Search_AStar → Algorithm A*
- Search_UCS → Algorithm UCS
- visualize_pacman_movement → Visualizes each action of Pacman in the given list of actions.
- clear_screen → Removes dots from the map after Pacman eats them.
- apply_action → Updates Pacman's current position on the map.
- print_map → Prints the map every time it's updated.

Summary of how to solve the requirements of the Pacman

Test.py

run_algorithm

Test

Executes the program with input as the map name and the algorithm's name to run.

UCS pseudocode for Pacman

```
Function search_UCS(map_game) returns actions, cost
        frontier ←  a priority queue with state, action, and order by cost
        explored ← an empty set
        start_state ← get start position of Pacman
        frontier ← put start_state with cost 0 and action empty
        Loop do
            if EMPTY?( frontier) then return failure
            cost, current_state, actions ← frontier.pop
            Add current_state to explored
            If not have nearest dot then return actions, cost
            If current_state is dot then
                Update mapgame
                Reset frontier with cost, current_state, actions
                Reset explored to empty set
            For each succ in mapgame.getSuccessor(current_state) do
                step_cost, next_state, direction = succ
                If next_state not in explored and not in frontier then
                    gn' ←  gn + step_cost
                    Add direction to list actions
                    Put (gn', next_state, new_action) to frontier
```

A* pseudocode for Pacman

Function manhattan_distance(state, map_game) **returns** distance:

goal_state ← find_nearest_goal_state(state, map_game)

If goal_state is none **return** 0

distance = abs(state.column - goal_state.colum) + abs(state.row - goal_state.row)

Function search_AStar(map_game) **returns** actions, cost

frontier ← a **priority queue** with fn, gn, state, action, and order by fn

explored ← an empty **set**

start_state ← get start position of Pacman

frontier ← put start state with fn = 0 + h(start state), gn= 0, action empty

Loop do

A* pseudocode for Pacman

```
Function search_Astar(map_game) returns actions, cost
    …………
    Loop do
        if EMPTY?( frontier) then return failure
        fn, gn, current_state, actions = frontier.pop
        Add current_state to explored
        If not have nearest dot then return actions, cost
        If current_state is dot then
            Update map game
            Reset frontier with h(current_state), gn, current_state, actions
            Reset explored to empty set
        For each succ in mapgame.getSuccessor(current_state) do
            step_cost, next_state, direction = succ
            If next_state not in explored and not in frontier then
                gn'← gn + step_cost
                Add direction to curent_state.list_actions
                fn = gn'+ h(current state)
                Put (fn, gn', next_state, new_action) to frontier
```

Propose one heuristic function

The Manhattan distance heuristic [2] estimates the total number of units of horizontal and vertical deviation between the current position of Pacman and the current position of the nearest food dot.

The admissibility property of the Manhattan heuristic $\Leftrightarrow \boldsymbol{h(n) \le h^*(n)}$ where

- $\boldsymbol{h(n)}$ is the Manhattan distance of state $\boldsymbol{n}$
- $\boldsymbol{h^*(n)}$ is the actual cost to reach the goal state.

This means:

- If $\boldsymbol{h(n) = 1}$, then $\boldsymbol{h^*(n)}$ must be at least 1 or greater.
- If $\boldsymbol{h(n) = 2}$, then $\boldsymbol{h^*(n)}$ must be at least 2 or greater.
- If $\boldsymbol{h(n) = 3}$, then $\boldsymbol{h^*(n)}$ must be at least 3 or greater.

Thus, for **every state n**, **we have $\boldsymbol{h(n) \le h^*(n)}$,** which satisfies **the admissibility property for the Manhanttan distance heuristic.**

Propose one heuristic function

The Manhattan distance heuristic estimates the total number of units of horizontal and vertical deviation between the current position of Pacman and the current position of the nearest food dot.

The consistency property of the Manhattan heuristic states that for ***every state n and n'*** (where n' is the next state of n, i.e., moving from n to n' costs 1 step), ***the estimated cost to reach the goal state from n plus the cost of moving from n to n' does not exceed the estimated cost to reach the goal state from n'.***

- $\boldsymbol{h(n)}$ is the Manhattan distance from state $\boldsymbol{n'}$ to goal
- $\boldsymbol{c(n, a, n')}$ is the cost of applying action $\boldsymbol{a}$ from state $\boldsymbol{n}$ to state $\boldsymbol{n'}$
- $\boldsymbol{h(n')}$ represents the Manhattan distance cost from state $\boldsymbol{n'}$ to goal

This means:

- If $\boldsymbol{h(n) = 1}$, then $\boldsymbol{h(n')}$ must be at least 0 or greater.
- If $\boldsymbol{h(n) = 2}$, then $\boldsymbol{h(n')}$ must be at least 1 or greater.
- If $\boldsymbol{h(n) = 3}$, then $\boldsymbol{h(n')}$ must be at least 2 or greater.

Therefore, for **every state $\boldsymbol{n}$,** we have $\boldsymbol{h(n) \leq c(n, a, n') + h(n')}$**,** which satisfies **the consistency property for the Manhattan distance heuristic.**

Comparison of A* and UCS Algorithms for Solving the Pacman Problem

ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

A* Algorithm	UCS Algorithm
is an **informed** search algorithm	is an **uninformed** search algorithm
If the **heuristic is consistent**, A* is **guaranteed** to find the optimal solution.	UCS **always finds** the optimal solution without the need for a heuristic
With a **good heuristic**, can **quickly find the optimal solution**. More **efficient** for **large state spaces.**	**Explores** the search space **uniformly**, which can lead to **higher time complexity**
higher space complexity, may need to store information about all nodes it has encountered	**lower space complexity**, only needs to store information about the nodes currently in the frontier

Conclusion: UCS generally **has a lower space complexity** compared to A*. A* is for **high-quality solutions with good heuristics** while UCS suits **optimal solutions in small spaces or low-cost exploration** scenarios.

References

[1] Thái Thanh Hải, “Data Structure & Algorithm - Graph Algorithms - Breadth First Search (BFS)” 2023. [Trực tuyến]. Địa chỉ: https://viblo.asia/p/data-structure-algorithm-graph-algorithms-breadth-first-search-bfs-gwd43kMM4X9. [Truy cập 27/2/2023].

[2] Blink, “Distance Measure trong Machine learning” 2021. [Trực tuyến]. Địa chỉ: https://viblo.asia/p/distance-measure-trong-machine-learning-ByEZkopYZQ0. [Truy cập 24/6/2021].

[3] Anshul Pareek, “How to solve 8 Puzzle problems using BFS and DFS and compare both in order to get optimal results?” 2023. [Trực tuyến]. Địa chỉ: https://www.linkedin.com/pulse/how-solve-8-puzzle-problems-using-bfs-dfs-compare-both-anshul-pareek. [Truy cập 20/09/2023].

[4] Gate Smashers, “A* algorithm in AI (artificial intelligence) in HINDI”, 2020. [Video]. Địa chỉ: https://www.youtube.com/watch?v=tvAh0JZF2YE. Truy cập [2020]

Group’s member

MSSV	Full name	Email
52200196	Nguyễn Quốc Duy	52200196@student.tdtu.edu.vn
52200188	Nguyễn Minh Nhựt	52200188@student.tdtu.edu.vn
52200193	Nguyễn Quang Trung	52200193@student.tdtu.edu.vn
52200154	Khưu Trùng Dương	52200154@student.tdtu.edu.vn

ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

Subject: Introduction to Artificial Intelligence

Thanks for your listening

This is the end of my group's presentation