Introduction to Artificial Intelligence

Faculty of DS & AI
Autumn semester, 2025

Trong-Nghia Nguyen

Business AI Lab

2025-08

I. Introduction

- 8-Puzzle is classic sliding puzzle with the objective is to reach a specific goal configuration by sliding tiles into the empty space

I. Introduction

- Real-word problem: Storage management

➢8 containers, 1 missing place

➢Each tile represents 1 containers, the empty tile represents missing place

I. Introduction

- This is a logistic problem that can be encountered in reality.

  Arranging the warehouse properly can affect:

1. Organization
2. Management
3. Coordinating the flow of goods

I. Introduction

- In this study, we apply three different algorithms:

1. BFS (Breadth-First Search)
2. GBFS (Greedy Best-First Search) with misplaced tile heuristic
3. A* search with Manhattan distance heuristic

→ Compare the effectiveness of three algorithms then point out the pros and cons of each search algorithm

II. Problem Fomulation

1. State space representation

State Space

1	3	2
4		5
6	8	7

Initial State

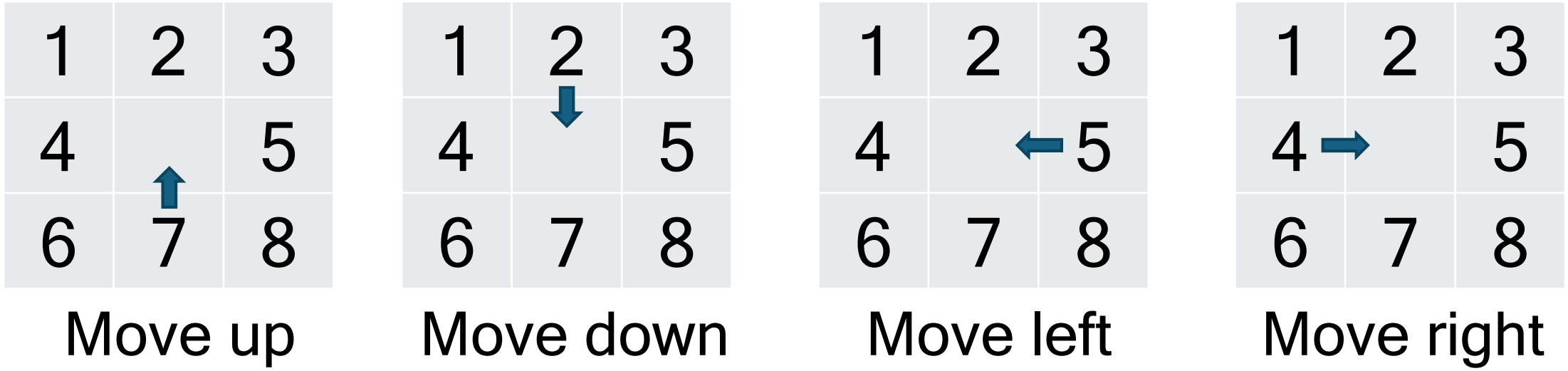
1	2	3
4	5	6
7	8	

Goal State

II. Problem Fomulation

1. State space representation

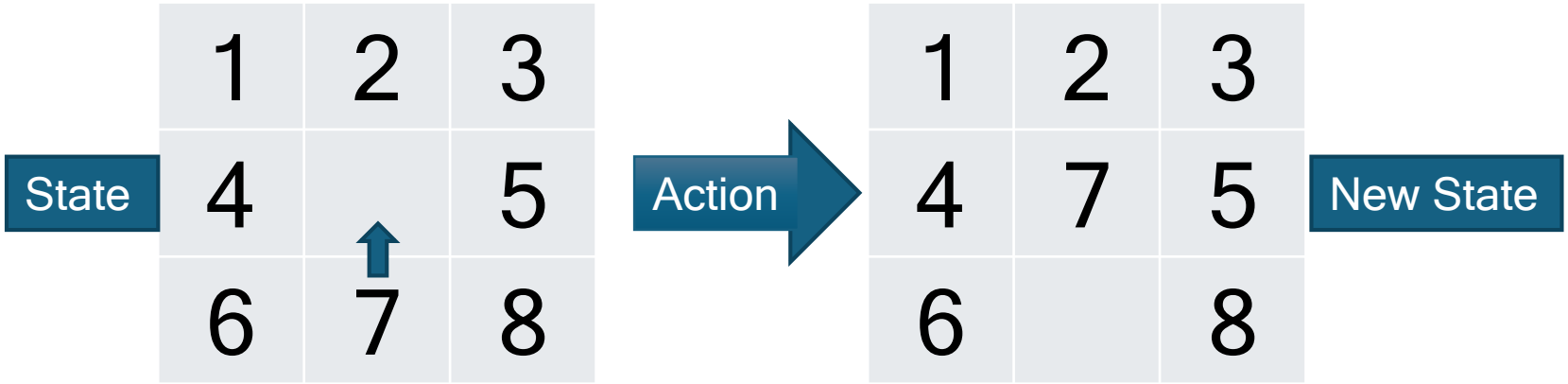
Actions: 2-4 actions possible depending on the position of the missing tile



II. Problem Fomulation

1. State space representation

- Transition Model: Given a state s and action a, the result is a new state s′ where the empty space and an adjacent tile have swapped positions.



- Path Cost: Each move has a uniform cost of 1, representing the effort required to relocate one tile.

II. Problem Fomulation

2. Heuristic Functions

Misplaced Tiles Heuristic (h1): The number of tiles not in their correct position (excluding the empty tile)

$$h_1(n) = \sum_{i=1}^{8} \begin{cases} 0 \text{ if } i \text{ is in correct position} \\ 1 \text{ if } i \text{ is misplaced} \end{cases}$$

II. Problem Fomulation

2. Heuristic Functions

Manhattan Distance Heuristic (h2): The Manhattan distances from each tile to its target position. The Manhattan distance between positions (x1,y1) and (x2,y2) is |x1−x2|+|y1−y2|.

$$h_2(n) = \sum_{i=1}^{8} |x_i - x_{goal,i}| + |y_i - y_{goal,i}|$$

III. Algorithms

1. Breadth-First Search (BFS)

BFS explores all nodes at depth d before exploring nodes at depth d + 1. It guarantees finding the optimal solution (shortest sequence of moves) but requires exponential space complexity.

```
Algorithm BFS(start, goal):
    openSet ← {start}
    visited ← ∅

    while openSet ≠ ∅ do
        current ← openSet.pop_front()
        if current = goal then
            return ReconstructPath(current)

        for each neighbor of current do
            if neighbor ∉ visited then
                visited.add(neighbor)
                openSet.push_back(neighbor)
    return failure
```

III. Algorithms

2. Greedy Best-First Search (GBFS)

GBFS expands the node that appears closest to the goal according to the heuristic function. While computationally efficient, it does not guarantee optimal solutions.

```
Algorithm GBFS(start, goal):
    openSet ← {start}
    while openSet ≠ ∅ do
        current ← node in openSet with lowest h(current)
        if current = goal then
            return ReconstructPath(current)

        openSet.remove(current)
        for each neighbor of current do
            if neighbor not yet visited then
                cameFrom[neighbor] ← current
                openSet.add(neighbor)
    return failure
```

III. Algorithms

3. A* Search

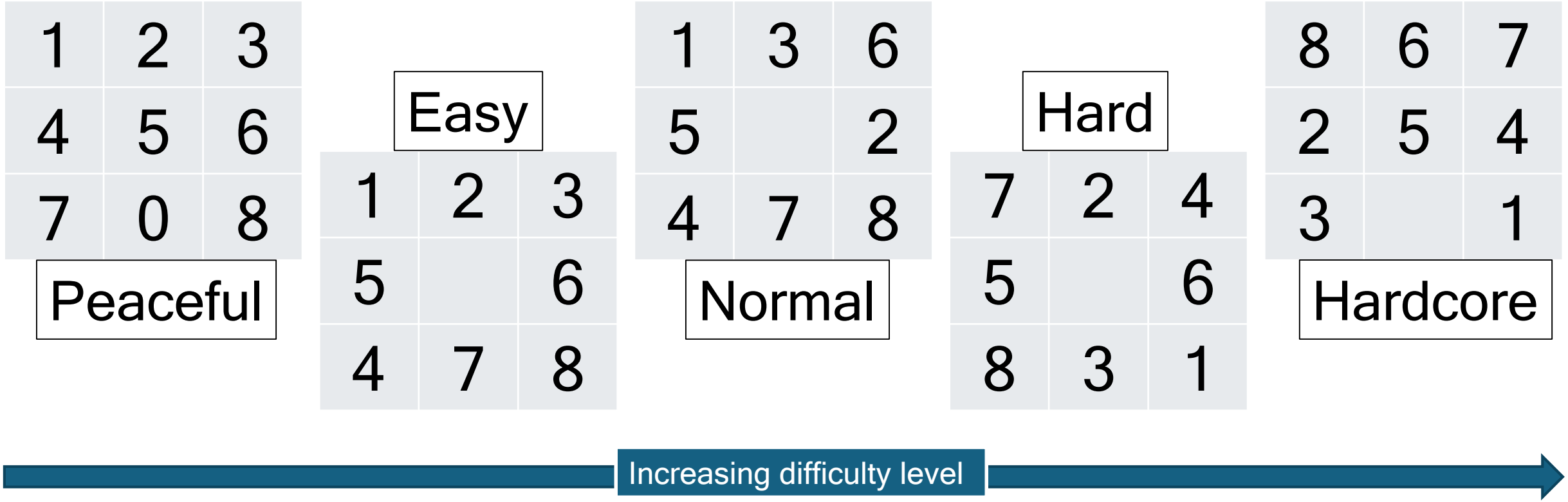
A* combines the advantages of BFS and GBFS by using an evaluation function f(n) = g(n)+h(n), where g(n) is the cost from the start to node n, and h(n) is the heuristic estimate from n to the goal.

```
Algorithm A*(start, goal):
    openSet ← {start}
    gScore[start] ← 0
    fScore[start] ← h(start)

    while openSet ≠ ∅ do
        current ← node in openSet with lowest fScore
        if current = goal then
            return ReconstructPath(current)

        openSet.remove(current)
        for each neighbor of current do
            tentative_gScore ← gScore[current] + d(current, neighbor)
            if tentative_gScore < gScore[neighbor] then
                cameFrom[neighbor] ← current
                gScore[neighbor] ← tentative_gScore
                fScore[neighbor] ← gScore[neighbor] + h(neighbor)
                if neighbor ∉ openSet then
                    openSet.add(neighbor)
    return failure
```

IV. Experiments & Results

1. Set up

There will be 5 levels to the experiment:



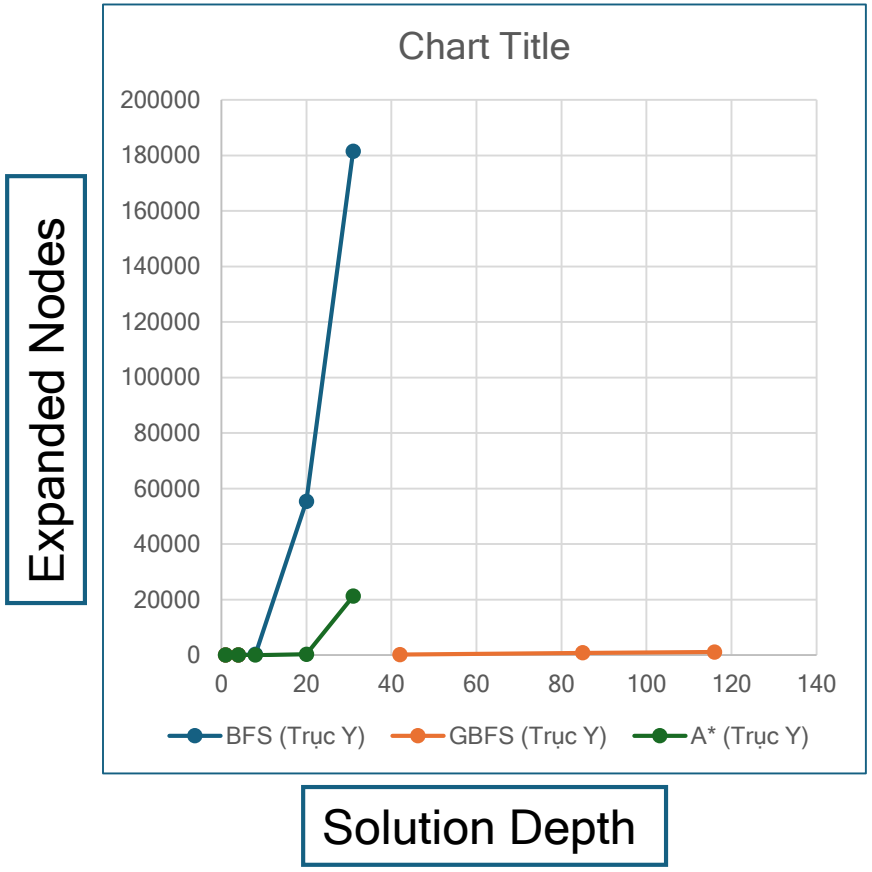
IV. Experiments & Results

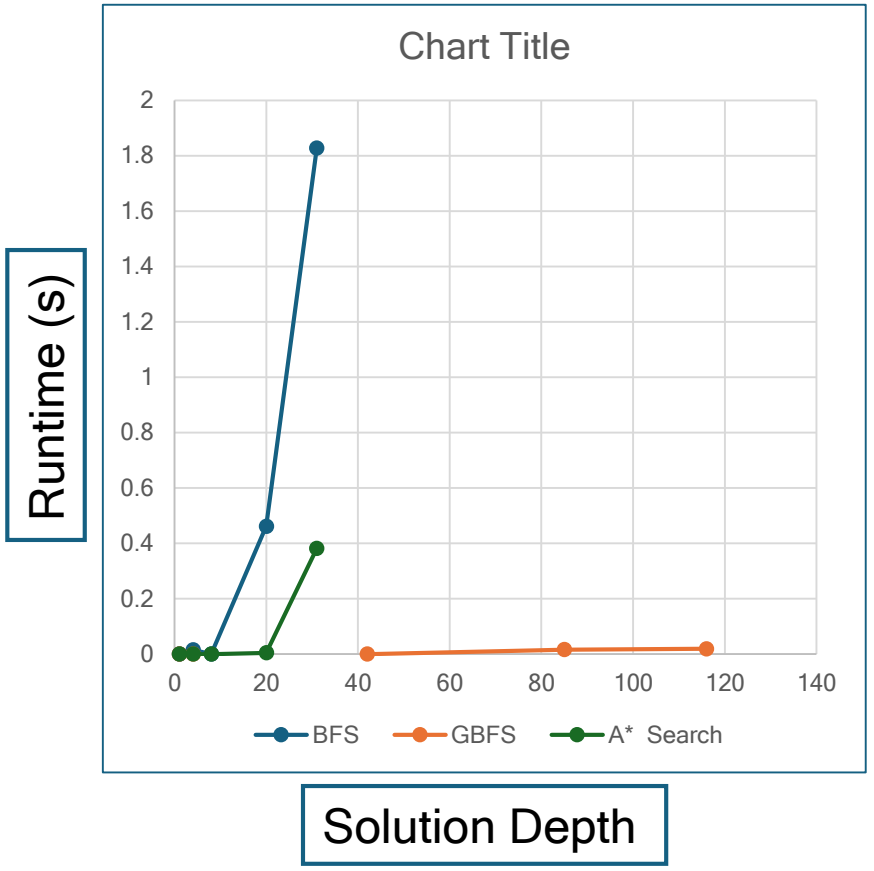
2. Performance Comparison

Test case	Algorithm	Solution Depth	Expanded Nodes	Runtime (s)
Peaceful	BFS GBFS A*	1 1 1	4 2 2	Very fast (approximately 0.0)
Easy	BFS GBFS A*	4 4 4	33 5 5	0.0156 0.0 0.0
Normal	BFS GBFS A*	8 42 8	311 206 13	0.0 0.0 0.0
Hard	BFS GBFS A*	20 116 20	55409 1134 283	0.4617 0.0192 0.0047
Hardcore	BFS GBFS A*	31 85 31	181440 834 21198	1.8268 0.0163 0.3813

IV. Experiments & Results

3. Visualization





V. Discussion

1. Algorithms Evaluation

BFS:

- Pros: ensuring finding optimal solution
- Cons: extremely ineffiecient for complicated case because it take a lot of memory usage and runtime
- Real-life applications: for situation in which optimal moveset is prioritized over short solving time

V. Discussion

1. Algorithms Evaluation

GBFS:

- Pros: fast, even for complex test case
- Cons: sometime find inefficient solution, leading to more 60-80% moves than necessary
- Real-life application: for scenario where quick response time is the most important

V. Discussion

1. Algorithms Evaluation

A* Search:

- Pros & Cons: balance between finding a good outcome and finding it fast, though sometimes cannot do both
- Real-life application: since it is a balanced search algorithm, it can be used in most scenario

V. Discussion

2. Real-world application insights

Optimal solution > Speed of finding solution

→BFS is ok, since it provides shortest way to arrange the storage, but computational cost and runtime can become excessively high

→A* search is a good search algorithm, since it is a balance search method, then it can be used in most scenario

V. Discussion

2. Real-world application insights

Optimal solution > Speed of finding solution

→GBFS alone does not seem like a suitable for storage management, because sometimes it bring non-optimal solution, which is really needed it storage management

V. Discussion

2. Real-world application insights

- Scalability Analysis: The exponential growth of BFS makes it unsuitable for storage with more than 15-20 containers requiring re-arrangement. GBFS scales linearly with problem complexity, while A* maintains reasonable performance even for complex scenarios

→ Hybrid Approaches: For practical implementation, a hybrid system could use GBFS for initial rapid solutions and A* for refinement when optimal arrangements are required

Thank you!

You're now ready to explore the exciting world of AI!