Introduction to Artificial Intelligence

Maze Pathfinding Comparison

Group 6
Mai Tuan Manh - 11247318
Mai Huy Dang - 11247269
Nguyen Huy Minh - 11247322

Business AI Lab

2025-10

Content

1. Introduction
2. Problem Formulation
3. Algorithms
4. Experiments & Results
5. Discussion

Content

Introduction

In the unfortunate case of **unexpected incidents** like fire or terrorism, safe and rapid evacuation becomes a matter of life and death.

=> People need to find the **nearest exit** to escape.

A typical example of a **state space search problem**, where the maze serves as a simplified model of the real-world scenario.

The goals of this project are:

- To model the evacuation problem as a maze search problem.
- To apply and compare search algorithms: **BFS, DFS, and GBFS**.
- To evaluate their performance and derive insights for real-world applications.

Large-scale events such as the A80 Museum have attracted huge crowds

Content

1. Introduction
2. **Problem Formulation**
3. Algorithms
4. Experiments & Results
5. Discussion

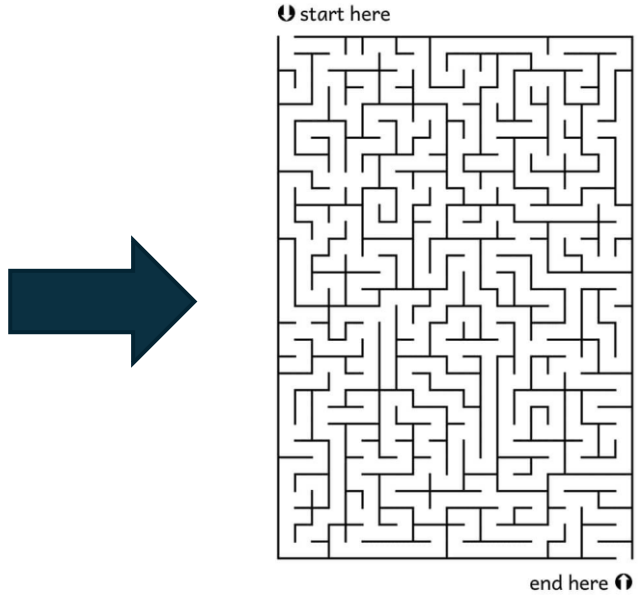
Problem Formulation

To simplify the real-world evacuation scenario, we model it as a **maze pathfinding problem.**

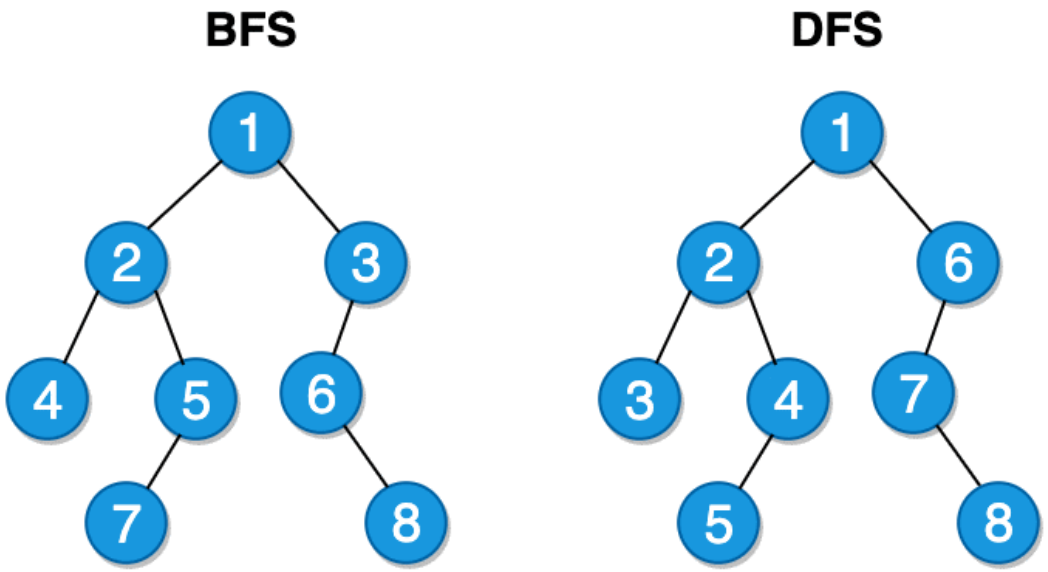
In this maze:

- Each cell represents a possible position.
- The **starting point (Start)** corresponds to the current location of a person.
- **Walls, booths, or obstacles** are represented as blocked cells.
- **Exits** represent the safe points to reach.
  The goal is to find the **shortest and safest path** from the start to the nearest exit.

Map of the 47th Scientific Conference on Dental Research



Problem Formulation



Apply **three classical search methods** to find and compare evacuation paths:

- **BFS (Breadth-First Search):** explores evenly
- **DFS (Depth-First Search):** explores deeply first
- **GBFS (Greedy Best First Search):** uses heuristic guidance

Since no real cost data is provided, the heuristic for GBFS is the **Manhattan Distance:**

$$\|\mathbf{x}_a - \mathbf{x}_b\|_M = |x_{a1} - x_{b1}| + |x_{a2} - x_{b2}|$$

Content

1. Introduction
2. Problem Formulation
3. **Algorithms**
4. Experiments & Results
5. Discussion

Algorithms

Create maze:

- The maze is generated using the **DFS Backtracking** method, producing a **perfect maze** — Exactly **one unique path** between any two points (A to B).
- To make the maze more realistic, some walls are **randomly removed**, creating an **imperfect maze**.
- In this type of maze, there may be **multiple possible paths** and even **loops** that return to previously visited points.
- This setup better **reflects real-world conditions** and allows us to more effectively **evaluate the strengths and weaknesses of each search algorithm**.

```python
# Create maze
import numpy as np
import random

def create_maze(dim, exits, i):
    random.seed(i)
    # Create a grid filled with walls ("#" = wall)
    maze = np.full((dim * 2 + 1, dim * 2 + 1), "#", dtype=str)
    # Start cell in logical grid coords (0,0)
    x, y = (0, 0)
    maze[2 * x + 1, 2 * y + 1] = "."
    # DFS backtracking stack
    stack = [(x, y)]
    while stack:
        x, y = stack[-1]
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        random.shuffle(directions)
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if (0 <= nx < dim and 0 <= ny < dim and
                maze[2 * nx + 1, 2 * ny + 1] == "#"):  # unvisited
                # Carve passage
                maze[2 * nx + 1, 2 * ny + 1] = "."
                maze[2 * x + 1 + dx, 2 * y + 1 + dy] = "."
                stack.append((nx, ny))
                break
        else:
            stack.pop()
```

Algorithms

We apply and compare three pathfinding algorithms:

DFS (Depth-First Search)

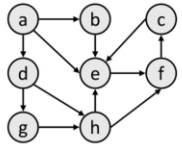
DFS pseudocode

function **dfs**(v_1, v_2):
  dfs(v_1, v_2, { }).

function **dfs**(v_1, v_2, *path*):
  path += v_1.
  mark v_1 as visited.
  if v_1 is v_2:
    a path is found!

  for each unvisited neighbor *n* of v_1:
    if dfs(*n*, v_2, *path*) finds a path: a path is found!

  path -= v_1. // path is not found.



- The *path* param above is used if you want to have the path available as a list once you are done.
  - Trace dfs(*a*, *f*) in the above graph.

2

GBFS (Greedy Best First Search)

```
Best-first search {

closed list = [ ]
open list = [start node]

    do {
            if open list is empty then{
                    return no solution
            }
            n = heuristic best node
            if n == final node then {
                    return path from start to goal node
            }
            foreach direct available node do{
                    if node not in open and not in closed list do {
                            add node to open list
                            set n as his parent node
            }
            delete n from open list
            add n to closed list
    } while (open list is not empty)
}
```

BFS (Breadth-First Search)

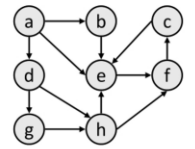
BFS pseudocode

function **bfs**(v_1, v_2):
  queue := {v_1}.
  mark v_1 as visited.

  while *queue* is not empty:
    v := *queue*.removeFirst().
    if *v* is v_2:
      a path is found!

    for each unvisited neighbor *n* of *v*:
      mark *n* as visited.
      queue.addLast(*n*).

  // path is not found.



- Trace bfs(*a*, *f*) in the above graph.

5

Algorithms

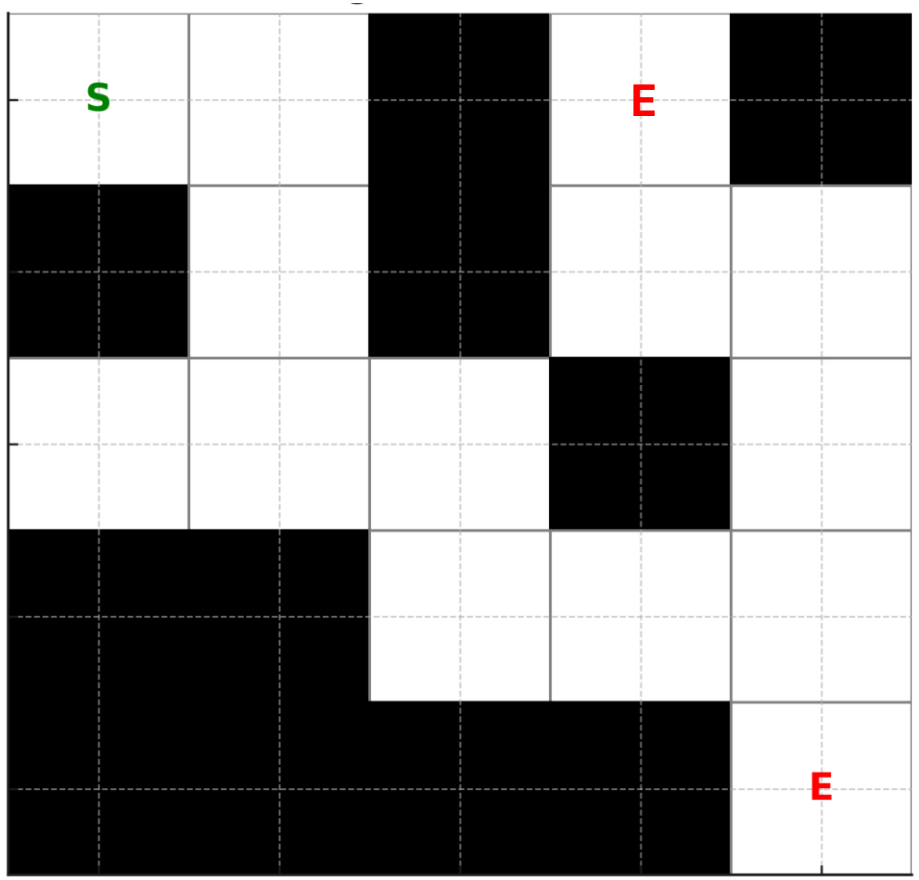
```python
# Testing multiple maze (benchmark)
import time
import tracemalloc

def benchmark(dim=3, exits=1, runs=10):
    results = { "BFS": {"times": [], "mems": []},
                "DFS": {"times": [], "mems": []},
                "GBFS": {"times": [], "mems": []} }

    for i in range(runs):
        # Generate a new maze with seed i
        grid = create_maze(dim, exits, i)

        for name, func in [("BFS", bfs), ("DFS", dfs), ("GBFS", gbfs)]:
            tracemalloc.start()
            time_start = time.time()
            _ = func(grid)
            time_end = time.time()
            _, peak = tracemalloc.get_traced_memory()
            tracemalloc.stop()
            results[name]["times"].append(time_end - time_start)
            results[name]["mems"].append(peak)

    # Print averages
    print(f"Benchmark results over {runs} mazes ({dim*2 + 1}x{dim*2 + 1}, {exits} exits):\n")
    for name, data in results.items():
        avg_time = sum(data["times"]) / len(data["times"])
        avg_mem  = sum(data["mems"]) / len(data["mems"])
        print(f"{name}: avg runtime {avg_time:.6f}s, avg peak memory {avg_mem/1024:.2f} KB")

    return results
```

A dedicated **benchmarking function** is used to test multiple maze instances and compute the **average runtime** and **peak memory usage** for each algorithm.

Content

1. Introduction
2. Problem Formulation
3. Algorithms
4. **Experiments & Results**
5. Discussion

Experiments & Results



Settings

Use the **DFS Backtracking algorithm** to generate mazes of different sizes, each maze contains **one starting point (S)** and **some exits (E).**

Three search algorithms — **DFS**, **BFS**, and **GBFS** — were tested.

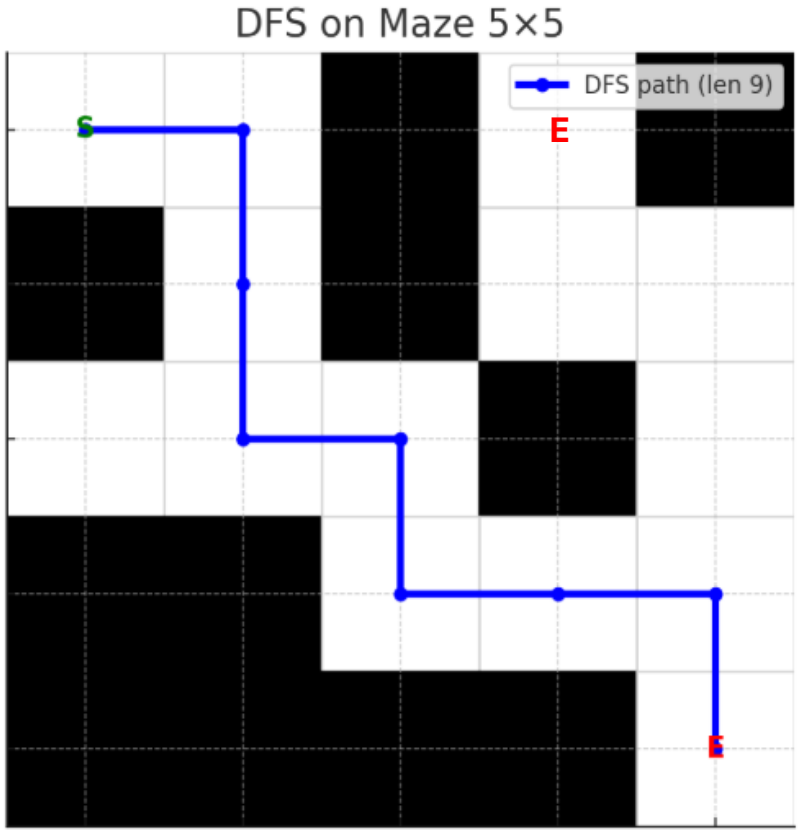
Each algorithm was executed **100 times on 100 different mazes**, and the **average runtime** and **peak memory usage** were recorded for comparison

Experiments & Results

Implementation steps

DFS

Step	Node	Neighbors	OPEN (Stack)
1	(0,0)	(0,1)	(0,1)
2	(0,1)	(1,1)	(1,1)
3	(1,1)	(2,1)	(2,1)
4	(2,1)	(2,0), (2,2)	(2,0), (2,2)
5	(2,0)	x (Dead-end)	(2,2)
6	(2,2)	(3,2)	(3,2)
7	(3,2)	(3,3)	(3,3)
8	(3,3)	(3,4)	(3,4)
9	(3,4)	(4,4) (end)	x

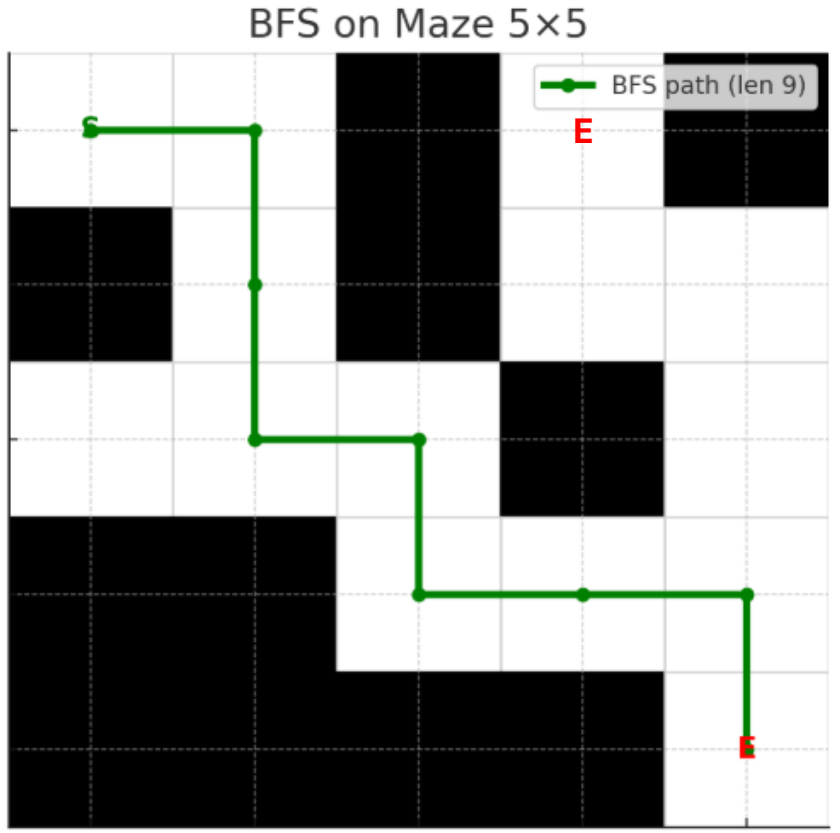


Experiments & Results

Implementation steps

BFS

Step	Node	Neighbors	OPEN (Stack)
1	(0,0)	(0,1)	(0,1)
2	(0,1)	(1,1)	(1,1)
3	(1,1)	(2,1)	(2,1)
4	(2,1)	(2,0), (2,2)	(2,0), (2,2)
5	(2,0)	x	(2,2)
6	(2,2)	(3,2)	(3,2)
7	(3,2)	(3,3)	(3,3)
8	(3,3)	(3,4)	(3,4)
9	(3,4)	(4,4) (end)	x

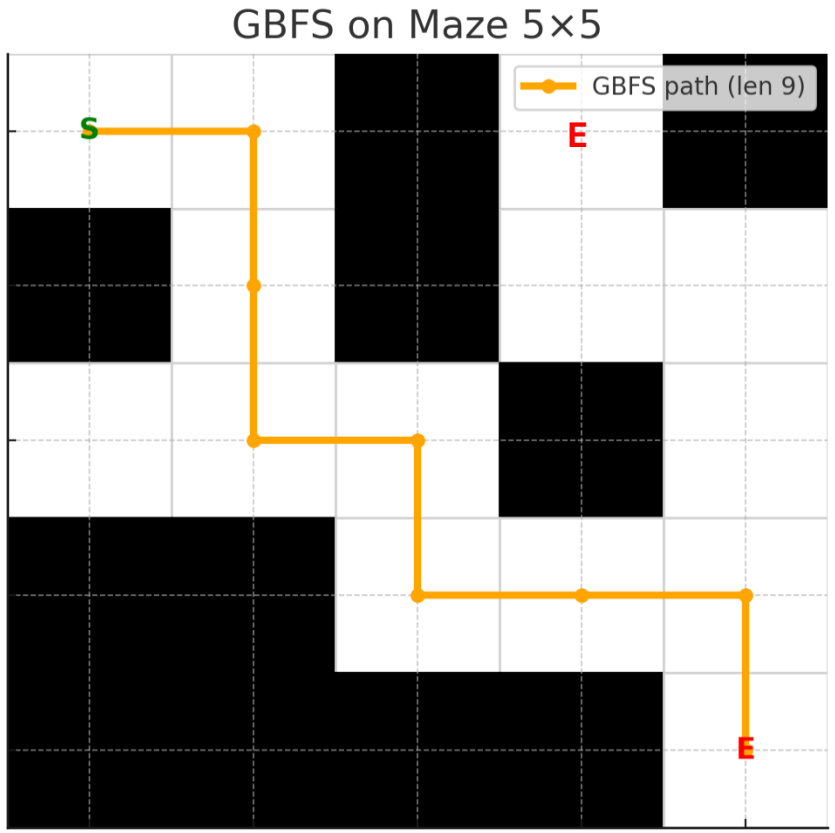


Experiments & Results

Implementation steps

GBFS

Step	Node	Neighbors	OPEN (Stack)
1	$(0,0)^3$	$(0,1)^2$	$(0,1)^2$
2	$(0,1)^2$	$(1,1)^3$	$(1,1)^3$
3	$(1,1)^4$	$(2,1)^4$	$(2,1)^4$
4	$(2,1)^4$	$(2,0)^5$, $(2,2)^3$	$(2,2)^3$, $(2,0)^5$
5	$(2,2)^3$	$(3,2)^3$	$(3,2)^3$
6	$(3,2)^3$	$(3,3)^2$	$(3,3)^2$
7	$(3,3)^2$	$(3,4)^1$	$(3,4)^1$
8	$(3,4)^1$	(4,4) (end)	x



Experiments & Results

Results comparison

```
result = benchmark(2, 2, 100)
```

```
Benchmark results over 100 mazes (5x5, 2 exits):

BFS: avg runtime 0.000229s, avg peak memory 2.24 KB
DFS: avg runtime 0.000254s, avg peak memory 1.76 KB
GBFS: avg runtime 0.000173s, avg peak memory 2.04 KB
```

```
result = benchmark(3, 2, 100)
```

```
Benchmark results over 100 mazes (7x7, 2 exits):

BFS: avg runtime 0.000390s, avg peak memory 2.71 KB
DFS: avg runtime 0.000480s, avg peak memory 2.65 KB
GBFS: avg runtime 0.000368s, avg peak memory 2.24 KB
```

In the **5×5** and **7×7** mazes, the results show similar patterns, with only **minor differences** in performance between BFS, DFS, and GBFS.

Experiments & Results

Results comparison

```
result = benchmark(100, 10, 100)
```

```
Benchmark results over 100 mazes (201x201, 10 exits):

BFS: avg runtime 0.230624s, avg peak memory 198.66 KB
DFS: avg runtime 0.265314s, avg peak memory 570.74 KB
GBFS: avg runtime 0.234619s, avg peak memory 111.66 KB
```

However, with larger mazes such as **201×201**, we can observe that:

- **BFS** still guarantees the shortest path but consumes more memory.
- **DFS** explores deeply, which often results in longer runtime and higher memory usage.
- **GBFS** while designed to be faster by using the Manhattan heuristic to guide the search, can sometimes become slower when multiple exits are present, since it must repeatedly compute heuristic values for each possible goal. (With a smarter and faster heuristic than Manhattan, GBFS could perform even better.)

Content

1. Introduction
2. Problem Formulation
3. Algorithms
4. Experiments & Results
5. **Discussion**

Discussion

Real-world terms

- If **maximum accuracy** is required, we should choose **BFS**, since it always finds the **absolutely shortest path**.

- In **emergency situations** such as evacuation, **GBFS** is a more suitable choice because it produces results **faster**, even though the path may not be perfectly optimal.

- For **DFS**, due to its “deep-first then backtrack” nature, it can be **very fast** if it happens to pick the right direction early — but if not, it may **wander inside the maze for a long time**.

Discussion

Practical Application

- Evacuation planning for large-scale events
- Guidance robots in seating areas
- Support booths for real-time evacuation route calculation

BFS is often combined with **GBFS (or A*)** because it ensures both **efficiency and speed**.

Heuristic functions are not only based on distance but also consider factors such as:

- **Smoke or heat levels** (in case of fire),
- **Crowd density** (to avoid congestion),
- **Door status** (open/locked).

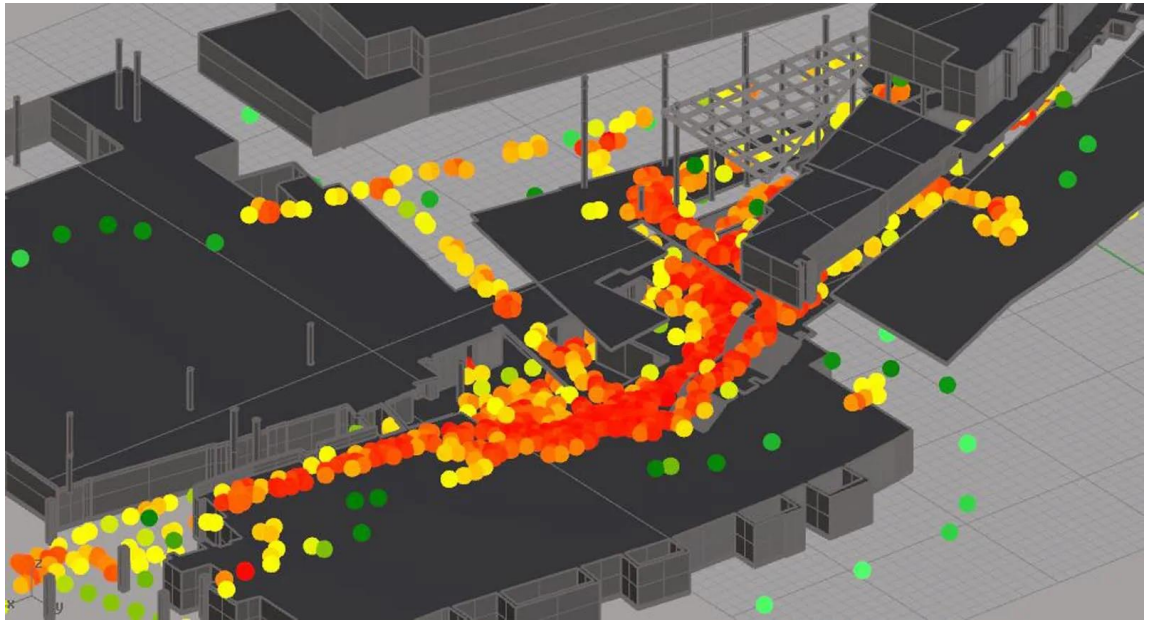
These methods can be integrated into an **Agent-Based Simulation (ABS)** model.

Discussion

Agent-Based Simulation

Example: Evacuation Simulation in Stadium A80 (5 exits)

- Each spectator is modeled as an **agent.**
- Each agent knows their **current position** and the **nearest exit direction.**
- **During the simulation:**
  - Each agent searches for an escape route (using BFS or GBFS).
  - When too many agents gather, **movement speed decreases** (congestion).
  - Some agents switch to **alternate exits** if they appear faster.
  - The system records: Average evacuation time, number of people successfully evacuated, congestion points.



*The **simulation results** provide insights for **real-world safety assessment** and help **optimize emergency exit designs**.*

Thank you!

We’re happy to answer any questions about our maze evacuation simulation and algorithm comparison.