Puzzle-8: Graph Search, IDS, and Uniform Cost Backtracking

Arpit Jain

September 21, 2025

A. Objective

The objective is to solve the 8-puzzle problem using a graph search agent. The goal is to implement:

- Graph search agent assuming uniform cost per move.
- Iterative Deepening Search (IDS).
- Backtracking to produce the solution path.
- Generation of solvable Puzzle-8 instances at a specific depth d.
- Measure memory and execution time for each instance.

B. Problem Statement

The 8-puzzle consists of 8 numbered tiles and a blank space arranged in a 3×3 grid. The goal is to move tiles until they reach the goal state:

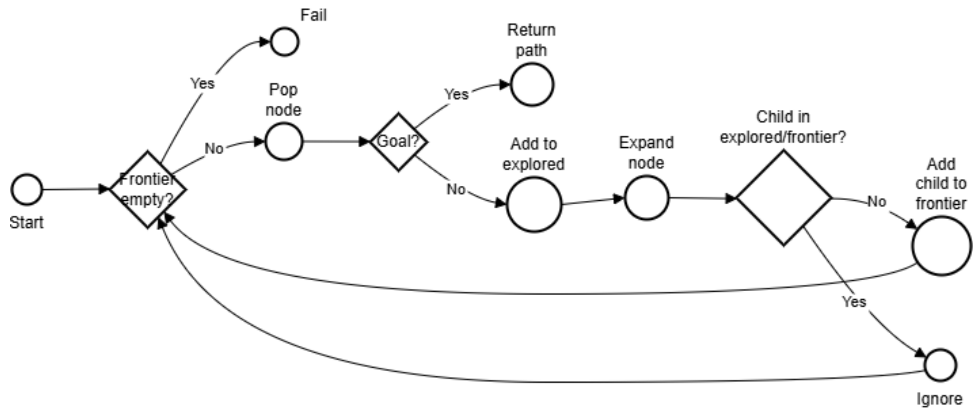
$$\text{Goal state: } [1, 2, 3, 4, 5, 6, 7, 8, 0]$$

where 0 represents the blank. Each move has equal cost. We are to implement graph search agents, generate instances at given depths, and record memory/time performance.

C. Methodology

- Each state is a list of 9 integers representing tiles row-wise.
- Successor states are generated by moving the blank in four directions.
- BFS implements uniform-cost search; IDS is used for memory-efficient deep search.
- Backtracking reconstructs the path from goal to start.
- Puzzle generation at depth d ensures solvable instances.

D. Flowchart



E. Algorithms

Algorithm 1 Graph Search Agent (Uniform-Cost / BFS)

```
Function: graph_search(initial_state, goal_state)
Initialize queue with initial_state and visited set
Initialize parent mapping: came_from[start] = None
while queue is not empty do
    Pop node from queue
    if node == goal_state then
        Backtrack using came_from to get path
        return path
    end if
    for each successor of node do
        if successor not in visited and not in queue then
            Add successor to queue
            came_from[successor] = node
        end if
    end for
end while
return failure
```

F. Results

G. Conclusion

The graph search agent successfully solves Puzzle-8 instances using uniform-cost search (BFS). IDS reduces memory requirements while guaranteeing completeness. Backtracking correctly reconstructs the solution path, and puzzle generation at depth d produces solvable instances. Memory, time, and moves vary with depth, demonstrating algorithm efficiency.

Algorithm 2 Iterative Deepening Search (IDS)

```
Function: IDS(start, goal)
Initialize depth limit d = 0
repeat
    Perform Depth-Limited Search with limit d
    if goal found then
        return path
    end if
    d ← d + 1
until goal found
return failure
```

Algorithm 3 Backtracking Path Retrieval

```
Function: backtrack(came_from, goal)
Initialize empty path list
current = goal
while current ≠ None do
    Append current to path
    current = parent from came_from
end while
return reversed path
```

Algorithm 4 Puzzle Generation at Depth d

```
Function: generate_puzzle_at_depth(d)
state = GOAL_STATE
for i = 1 to d do
    Generate successors of state
    Randomly select one successor as new state
end for
return state
```

Depth	Memory (MB)	Time (s)	Moves
6	0.0	0.0001	0
8	0.01	0.0002	2
10	0.0	0.0	0
13	0.24	0.0065	9
15	0.03	0.0015	7
17	0.03	0.0015	7
20	0.01	0.0004	4
25	0.08	0.0043	9
40	0.01	0.0004	4

Table 1: Memory and time requirements for solving Puzzle-8 instances at depth d.