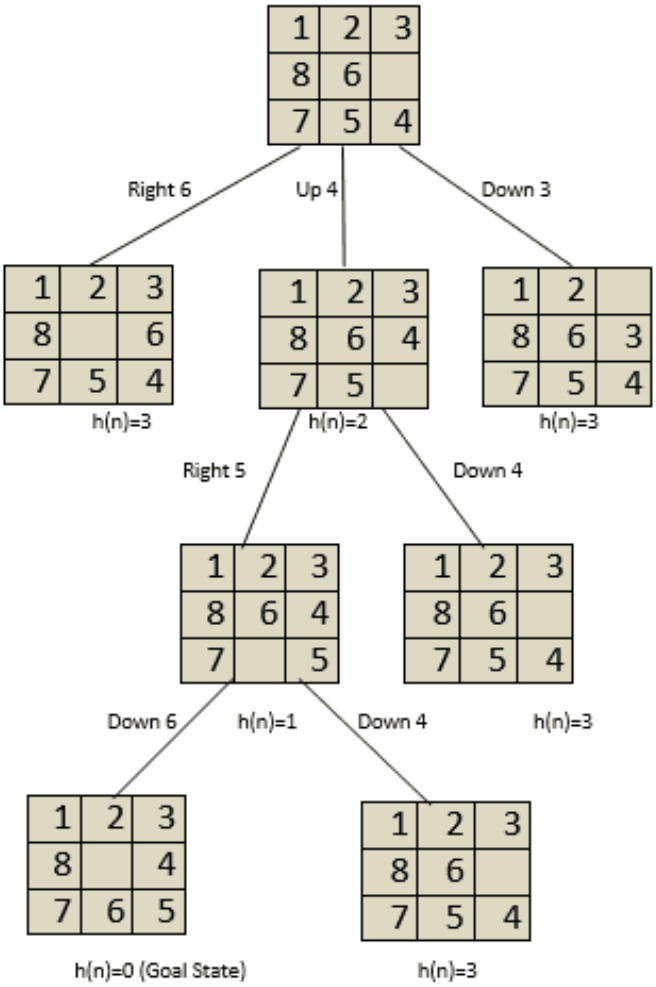


8 puzzle A.I

INFORMED & UNINFORMED SEARCH

7427 | 7806 | 7861

Introduction

The 8 Puzzle Game is a classic problem in artificial intelligence and computer science that involves sliding tiles within a 3x3 grid to achieve a specific arrangement, typically from a scrambled initial state to a goal state. Various algorithms can be employed to solve this puzzle efficiently. In this report, we analyze the implementation of using informed and uninformed search like Breadth-First Search (BFS), Depth-First Search (DFS), and A* Search algorithms for solving the 8 Puzzle Game.

Breadth-First Search (BFS) Algorithm

The BFS algorithm is a graph traversal technique that explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. In the provided code, the BFS algorithm starts by initializing the frontier as a deque(queue), storing nodes to be explored. It iteratively explores nodes at each level, maintaining a set of explored states to avoid duplication. If the goal state is found, the algorithm backtracks to construct the path from the initial state to the goal state. Performance metrics such as path length, explored nodes, search depth, and execution time are recorded.

Depth-First Search (DFS) Algorithm

In contrast to BFS, the DFS algorithm explores as far as possible along each branch before backtracking. The DFS implementation in the code also utilizes a deque(stack) for the frontier and keeps track of explored states. Similar to BFS, it records performance metrics and constructs the path upon finding the goal state.

Class Node importance

We implemented a node class to help us in optimizing the running time of the code in getting the path of the solution, This class is only used in BFS and DFS specially needed in DFS because of the stack implementation and the need of finding the parent of the goal node, we store the node in the path array with node on the parent.

A* Search Algorithm

A* Search is an informed search algorithm that uses heuristic evaluation to guide the search process efficiently. The implementation includes two heuristic functions: Manhattan distance and Euclidean distance. A priority queue is employed to prioritize nodes based on the sum of the heuristic value and the cost to reach the current state. The algorithm maintains a parent-child relationship to reconstruct the optimal path upon reaching the goal state.

Solvability Check

Before applying the algorithms, the code checks the solvability of the initial puzzle state using inversion counting. If the puzzle is unsolvable, an appropriate message is displayed.

Performance Analysis

Breadth-First Search (BFS)

Advantages:

- Guarantees the shortest path to the goal state due to its breadth-first nature.
- Suitable for solving puzzles with a finite state space and uniform cost edges.
- Optimal and complete

Disadvantages:

- Requires substantial memory to store the frontier, especially for puzzles with a large state space.
- May explore many unnecessary nodes at deeper levels before finding the goal state.

Depth-First Search (DFS)

Advantages:

- Memory-efficient as it only needs to store the path from the root to the current node.
- Suitable for puzzles with deep states and limited branching factor.

Disadvantages:

- Not complete, Not optimal as it may find a solution that is not the shortest path.
- Can get stuck in infinite loops if the search space contains cycles.

A* Search

Advantages:

- Complete, Provides an optimal solution if an admissible heuristic is used.
- Efficiently explores promising paths based on heuristic estimates.

Disadvantages:

- Requires careful selection of heuristic functions to ensure admissibility and accuracy.
- May suffer from increased computational complexity if the heuristic is not well-designed.

Recommendations

Algorithm Selection:

- Choose BFS when optimality and short solution paths are crucial.
- Consider DFS for memory-constrained environments or problems with deep states.
- Prefer A* Search for problems where heuristic estimates can guide the search efficiently.

Heuristic Design:

- Ensure heuristic functions used in A* Search are admissible and consistent to guarantee optimality and efficiency.
- Experiment with different heuristic functions to find the most suitable one for the problem domain.

Memory Management:

- Optimize memory usage by implementing space-saving data structures or pruning techniques, especially for BFS.
- Consider dynamic memory allocation and deallocation to reduce memory overhead during the search.

Parallelization:

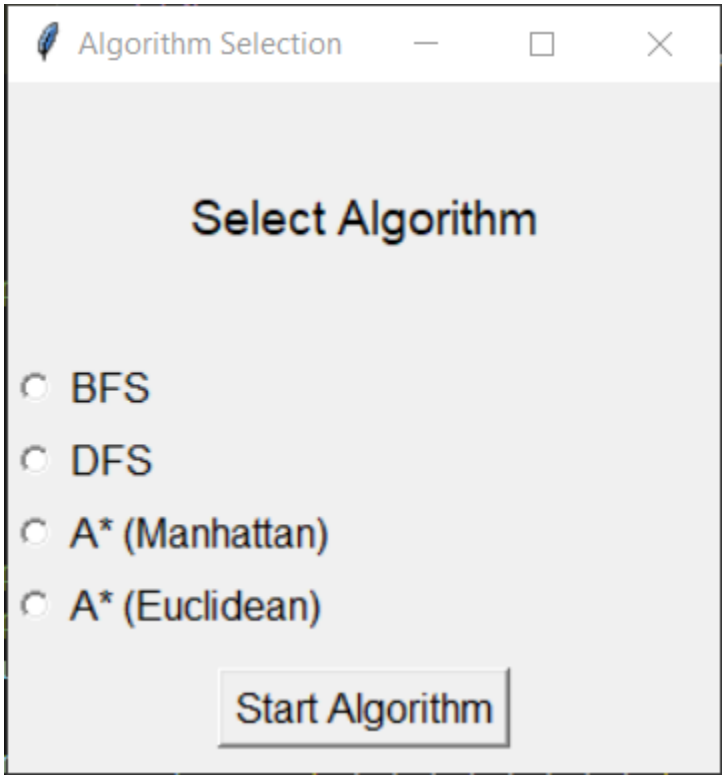
Explore parallel processing techniques to speed up the search process, particularly for A* Search where node evaluations can be parallelized.

Conclusion

The code we wrote offers efficient solutions to the 8 Puzzle Game using BFS, DFS, and A* Search algorithms. It incorporates solvability checking and provides a user-friendly interface for interactive exploration of the solving process. The algorithms' performance can be further evaluated and compared using various initial puzzle configurations. Overall, the implementation demonstrates effective problem-solving strategies in artificial intelligence.

Sample runs.

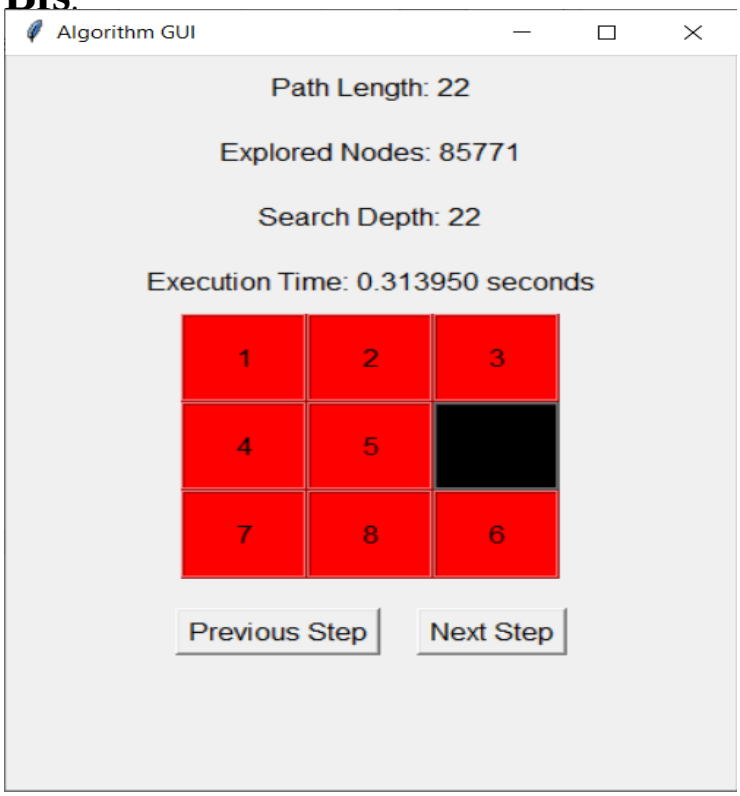
First we choose algorithm :



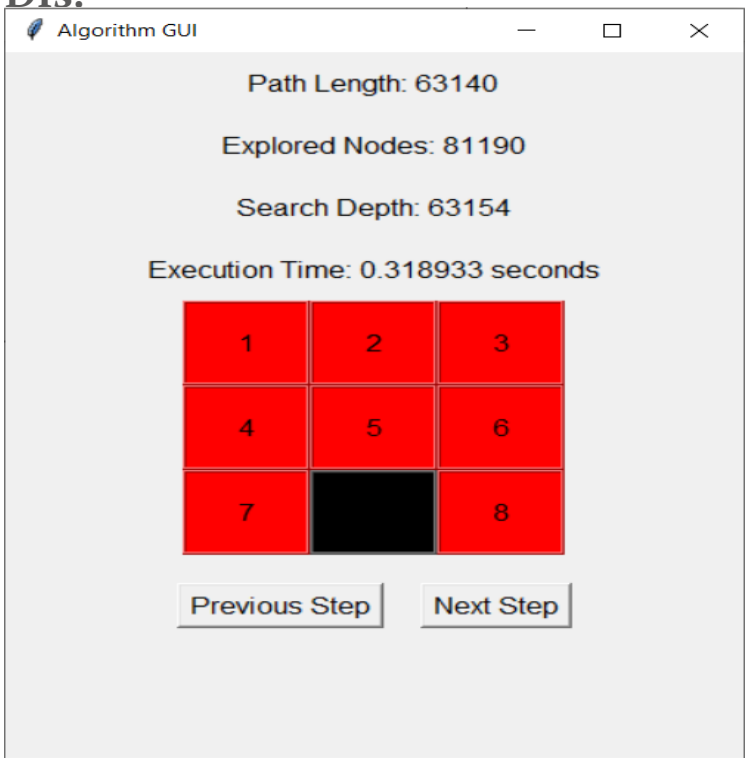
For this sequence:

1	2	3
4	5	6
7	8	0

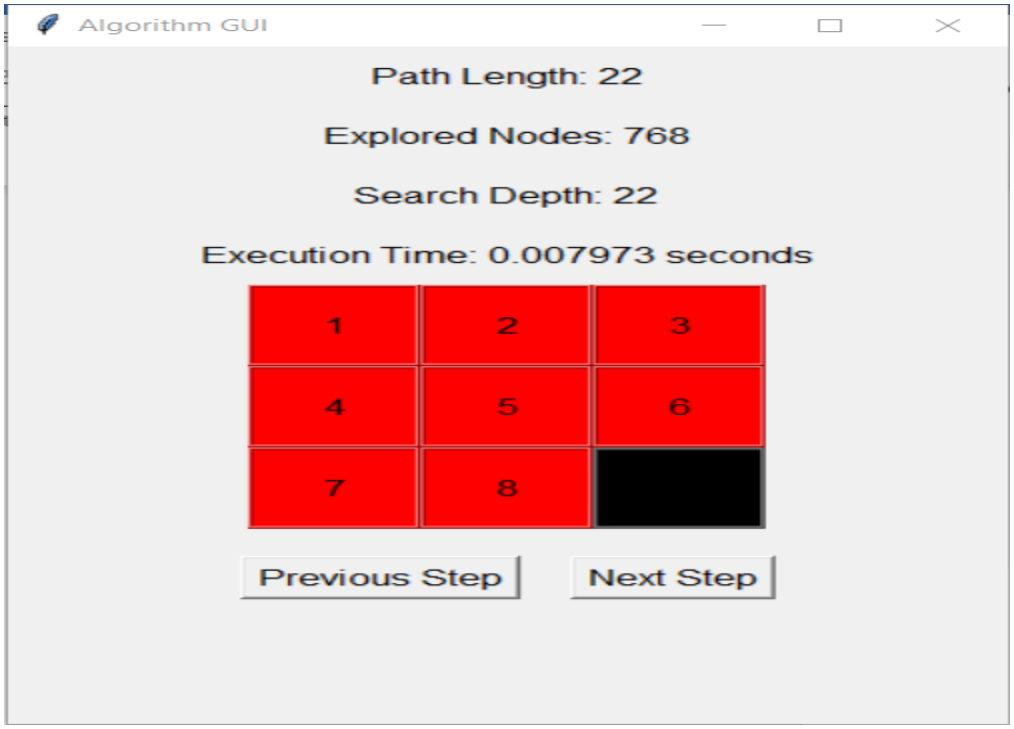
Bfs:


Algorithm GUI
Path Length: 22
Explored Nodes: 85771
Search Depth: 22
Execution Time: 0.313950 seconds
1
2
3
4
5
7
8
6
Previous Step
Next Step

Dfs:


Algorithm GUI
Path Length: 63140
Explored Nodes: 81190
Search Depth: 63154
Execution Time: 0.318933 seconds
1
2
3
4
5
6
7
8
Previous Step
Next Step

A* Manhattan



A* Euclidean: