A Project On:

8-Puzzle Problem

Submitted To:

Faculty Of Computer Science and Information Technology

Submitted By:

Abdallah Elsawy

Ahmed Mostafa

Abdelrahman Mohamed

Mohamed Hamdy

Abdallah Elmelegy

Supervised By: Eng. Omar Zantout

8-Puzzle Problem – Problem Overview

- The 8-Puzzle Problem is a classic problem in Artificial Intelligence used to evaluate and compare different state-space search algorithms. It consists of a 3×3 grid containing eight numbered tiles (1–8) and one empty space. A move is performed by sliding a tile adjacent to the empty space into it.
- The objective of the problem is to transform a given initial configuration into a predefined goal configuration using the minimum number of valid moves, following the movement constraints of the puzzle.

➔ Problem Characteristics

- **State Space**: Each configuration of the puzzle represents a unique state.
- **Actions**: Move the blank tile up, down, left, or right (when valid).
- **Goal Test**: Check whether the current state matches the goal state.
- **Path Cost**: Each move has a uniform cost (cost = 1).

➔ Why the 8-Puzzle Problem?

➔ The 8-puzzle is widely used because:

- It has a large but finite state space, making it suitable for search algorithm comparison.
- It clearly demonstrates the trade-offs between time complexity, space complexity, optimality, and completeness.
- It supports both uninformed (blind) and informed (heuristic-based) search strategies.

➔ Project Focus

This project focuses on solving the 8-Puzzle Problem using multiple AI search algorithms and comparing their performance in terms of:

- Solution optimality
- Memory usage
- Execution time
- Ability to handle complex initial states

By applying different algorithms to the same puzzle instances, the project highlights the strengths and limitations of each approach and demonstrates why heuristic-based algorithms, such as A*, are generally more efficient for this type of problem.

Breadth First Search (BFS) Algorithm To Solve The 8- Puzzle Problem:-

1. BFS Overview

- Breadth-First Search is an uninformed search algorithm that:

- Explores all states level by level, ensuring shallower nodes are expanded first.
- Uses a queue (FIFO) data structure.
- Guarantees finding the shortest solution if one exists.
- Requires more memory than DFS because it stores all nodes at the current depth.

➔ **For the 8-puzzle, BFS requires:**

- A visited set to avoid revisiting states.
- A queue to manage the frontier of unexplored states.

2. State Representation

- Each board configuration is treated as a node in the state-space.
- Represent each state as a tuple of 9 elements.
- 0 denotes the empty tile.

Example:
[5, 6, 8, 0,4,7,1, 3, 2]

3. How BFS Solves the 8-Puzzle

BFS solves the 8-puzzle by exploring all board configurations level by level. The algorithm ensures that the shortest path to the goal is found.

Process:

1. Push the initial puzzle configuration into a queue.
2. Remove the front state from the queue and check if it matches the goal state.
3. If it is not the goal, generate all valid successor states by moving the blank tile.
4. Add each successor state to the end of the queue, ensuring they have not been visited

before.

5. Repeat until the goal state is reached or the queue is empty.

To prevent infinite loops, a visited set keeps track of already explored states. BFS continues until the goal is found, guaranteeing the shortest sequence of moves.

4. Performance

4.1 Time Complexity

- Worst-case: O(b^d)
- b = branching factor
- d = depth of the shallowest solution
- BFS may explore many nodes but ensures the shortest solution.

4.2 Space Complexity

- O(b^d)
- BFS stores all nodes at the current depth

4.3 Optimality

- BFS is optimal for unweighted moves.
- Always finds the solution with the minimum number of moves.

4.4 Completeness

- BFS is complete; it will find a solution if one exists.

5. Advantages and Disadvantages Advantages:

- Guarantees the shortest solution.
- Simple and systematic exploration.
- Complete for finite problems.

Disadvantages:

- High memory consumption.
- Slower than DFS for large depth or branching factor.
- Less practical for very large state spaces.

6. Conclusion

Breadth-First Search reliably solves the 8-puzzle, guaranteeing the shortest solution. Its systematic exploration ensures completeness and optimality. However, high memory usage makes it less suitable for larger or more complex puzzles. For more scalable solutions, heuristic searches like A* are preferred.

1.Depth-First Search (DFS) Overview

Depth-First Search is an uninformed search algorithm that:

- Explores one branch of the search tree as deeply as possible before backtracking.
- Uses a stack (LIFO) data structure.
- Does not guarantee the shortest solution.
- Has low memory usage compared to Breadth-First Search.

For the 8-puzzle, DFS must include:

- A visited set to avoid infinite loops
- A depth limit to prevent infinite descent

---

2. State Representation

- Each state is represented as a tuple of 9 elements
- 0 represents the empty tile

Example:

(1, 2, 3,

4, 0, 6,

7, 5, 8)

---

3. How DFS Solves the 8-Puzzle

Depth-First Search solves the 8-puzzle by treating each board configuration as a node in a state-space tree. The algorithm starts from the initial state and explores one possible sequence of moves as deeply as possible before backtracking.

The process works as follows:

1. The initial puzzle configuration is pushed onto a stack.
2. The algorithm pops the top state from the stack and checks whether it matches the goal state.
3. If it is not the goal, all valid successor states are generated by moving the blank tile (up, down, left, right).
4. These successor states are pushed onto the stack, extending the current path.
5. DFS continues expanding states along a single path until:
    - The goal state is found, or
    - A predefined depth limit is reached.
6. When a path cannot be expanded further, the algorithm backtracks and explores alternative paths.

To avoid infinite loops, a visited set is used so that previously explored states are not revisited. A depth limit ensures that the algorithm terminates even if the solution is very deep or does not exist.

---

4. Performance Analysis

4.1 Time Complexity

- Worst-case time complexity: O(b^d)
    - b = branching factor (≤ 4 for 8-puzzle)
    - d = depth limit
- DFS may explore many unnecessary states before finding a solution

4.2 Space Complexity

- O(b × d)
- Much lower memory usage compared to BFS
- Only stores the current path and visited states

4.3 Optimality

- DFS does not guarantee the shortest path
- The first solution found may be suboptimal

4.4 Completeness

- Not complete without a depth limit
- Becomes complete with Depth-Limited DFS, but may miss deeper solutions

---

5. Advantages and Disadvantages

Advantages

- Simple to implement
- Low memory consumption
- Useful when solution depth is small

Disadvantages

- May get stuck exploring deep paths
- Not optimal
- Sensitive to depth limit choice

6. Conclusion

Depth-First Search can solve the 8-puzzle problem but is not ideal for finding optimal solutions. Its low memory usage makes it suitable for demonstration purposes and small search depths. For real-world applications, informed search algorithms such as A* are preferred.

This implementation demonstrates a clear understanding of:

- State-space representation
- DFS mechanics
- Performance trade-offs

- **Uniform Cost Search (UCS) Overview**
- **Uniform Cost Search is an uninformed search algorithm that:**

- Expands the node with the lowest cumulative path cost first.
- Uses a priority queue (min-heap) to select the next node.
- Guarantees finding the optimal solution when path costs are non-negative.
- Can have high memory usage for large search spaces.

For the 8-puzzle, UCS ensures that the shortest sequence of moves is found.

---

2. State Representation

- Each state is represented as a tuple of 9 elements
- 0 represents the empty tile

Example:

(1, 2, 3,

4, 5, 0,

6, 7, 8)

---

3. How UCS Solves the 8-Puzzle

Uniform Cost Search treats each puzzle configuration as a node in a state-space tree. The algorithm works as follows:

1. The initial state is pushed into a priority queue with a cumulative cost of 0.
2. UCS repeatedly pops the state with the lowest cumulative cost.
3. If this state matches the goal, the solution is returned.
4. Otherwise, all valid successor states are generated by sliding the empty tile.
5. Successor states not yet visited are pushed into the priority queue with an incremented cost.
6. The process continues until the goal is found or all states are explored.

Each move has a uniform cost of 1, so UCS effectively finds the shortest sequence of moves to reach the goal.

4. Performance Analysis

4.1 Time Complexity

- Worst-case: O(b^d)
    - b = branching factor (≤4)
    - d = depth of the optimal solution

4.2 Space Complexity

- O(b^d)
- Stores all generated states in the priority queue and visited set

4.3 Optimality

- UCS guarantees the shortest path when all move costs are equal and non-negative

4.4 Completeness

- UCS is complete for finite state space

---

5. Advantages and Disadvantages

Advantages

- Finds the optimal solution
- Complete for finite problems

Disadvantages

- High memory usage for large search spaces
- Can be slower than informed search algorithms like A*

6. Conclusion

Uniform Cost Search is a reliable method to solve the 8-puzzle optimally. It expands nodes by cumulative cost, ensuring the shortest sequence of moves is found. While memory-intensive, it is a good demonstration of uninformed optimal search in AI.

This implementation demonstrates a clear understanding of:

- State-space representation
- UCS mechanics
- Performance trade-offs

iterative Deepening Search (IDS) for the 8-Puzzle Problem

Algorithm Overview

Iterative Deepening Search (IDS) is a search algorithm that combines the advantages of Depth-First Search (DFS) and Breadth-First Search (BFS).
It performs a series of depth-limited DFS searches, where the depth limit increases gradually until the goal state is found.

In the context of the 8-Puzzle problem, IDS explores possible tile movements level by level while maintaining low memory usage.

Why Use IDS in the 8-Puzzle?

The 8-Puzzle has a large state space, and using BFS may consume excessive memory, while DFS may go too deep without finding the solution.
IDS solves this by:

- Limiting depth to avoid infinite paths.
- Gradually increasing the depth to ensure completeness.
- Guaranteeing the shortest solution path (optimal solution).

How IDS Solves the 8-Puzzle?

- **Algorithm (Step by Step): Iterative Deepening Search for 8-Puzzle**

- **Step 1: Represent the Puzzle Initial State**

- The puzzle is represented as a 3×3 grid (or 1D array) containing numbers from 1 to 8 and one empty tile represented by 0.
  This state is considered the starting point of the search.

- **Step 2: Find the Empty Tile (0)**

- The algorithm locates the position of the empty tile (0) in the current puzzle state.
  This position determines which moves are possible.

- **Step 3: Determine Valid Moves (UP – DOWN – LEFT – RIGHT)**

- Based on the position of the empty tile, the algorithm determines the valid moves without going outside the puzzle boundaries.
  Only legal moves are considered for state expansion.

- **Step 4: Generate a New State (Apply One Move)**

- A new puzzle state is generated by swapping the empty tile with an adjacent tile according to a valid move.
  Each move results in a new child state in the search tree.

- **Step 5: Goal Test**

- The generated state is compared with the predefined goal state.
  If both states are identical, the solution has been found and the search stops.

- **Step 6: Apply Depth-Limited Search (DLS)**

- The algorithm performs a Depth-First Search but stops expanding nodes when the current depth limit is reached.
  This prevents the search from going too deep in unpromising paths.

- **Step 7: Apply Iterative Deepening Loop**

- The depth limit starts from 0 and is increased gradually (0, 1, 2, …).
  At each iteration, DLS is restarted from the initial state until the goal state is found.

- **Step 8: Output the Solution**

- Once the goal is reached, the algorithm outputs the solution details.

Properties of Iterative Deepening Search (IDS)

Completeness:

Iterative Deepening Search is guaranteed to find a solution if one exists, because it systematically explores all depths starting from zero and increases the depth limit gradually.

Optimality:

IDS always finds the shortest solution path since it explores the search space level by level, similar to Breadth-First Search.

Time Complexity:

The time complexity of IDS is (O(b^d)), where (b) is the branching factor and (d) is the depth of the optimal solution.

Space Complexity:

The space complexity is (O(b \times d)), which is significantly lower than Breadth-First Search because IDS stores only a single path at a time.

Advantages

- Uses significantly less memory than BFS.
- Guarantees optimal solutions like BFS.
- Suitable for problems with unknown solution depth.

Disadvantages

- Repeated exploration of nodes at each depth level.
- Can be slower than heuristic-based algorithms like A*.

Conclusion

In the 8-Puzzle problem, Iterative Deepening Search provides a strong balance between memory efficiency and optimality.
While it may perform redundant computations, it remains a reliable and practical algorithm when memory.

A* Search For Solving 8-Puzzle Problem (With Manhattan Distance Heuristic)

Algorithm Overview:

*A** search using the Manhattan distance heuristic is a pathfinding algorithm that finds the optimal solution to problems by combining actual cost so far with an estimated cost to reach the goal. The Manhattan distance heuristic measures how far each tile is from its goal position by counting horizontal and vertical moves, making it an admissible and efficient guide for the search.

Why use A* Search in the 8-Puzzle?

The 8-Puzzle has a large state space, but A* search balances optimality with efficiency

How A* Search solves the 8-Puzzle Problem?

1. **Initialize frontier**
   - **Action:** Push the start state with $f = h(\text{start})$, $g = 0$, and an empty path.
   - **Reason:** Start exploring from the initial configuration, prioritized by heuristic.
2. **Initialize explored**
   - **Action:** Create an empty set to track visited states (by tuple key).
   - **Reason:** Avoid revisiting states and looping.
3. **Main loop: pop best candidate**
   - **Action:** Pop the state with lowest f from the heap.
   - **Check goal:** If it matches the goal, return path + [state].
   - **Mark explored:** Add the state's tuple key to the explored set.
4. **Expand neighbors**
   - **Action:** Generate all valid neighbor states by sliding the blank.
   - **For each neighbor:**

        - **Skip explored:** If its tuple key is in explored, continue.
        - **Compute costs:** New $g' = g + 1$; $h' = h(\text{neighbor})$.
        - **Push to frontier:** With priority $f' = g' + h'$, and extend path with the current state.

5. **Termination**
    - **Success:** When a popped state equals the goal, return the sequence of states from start to goal.
    - **Failure:** If the frontier empties without reaching the goal, return None (unsolved/unsolvable start).

Properties of A* Search

1. Completeness

- A* is **complete**, meaning it will always find a solution if one exists, provided the branching factor is finite and step costs are positive.

2. Optimality

- If the heuristic function $h(n)$is **admissible** (never overestimates the true cost to reach the goal), A* guarantees the **optimal solution** (shortest path or minimum cost).

3. Optimal Efficiency

- Among all optimal algorithms using the same heuristic, A* is **optimally efficient**.
- This means no other algorithm will expand fewer nodes than A* before finding the optimal solution.

4. Evaluation Function

- A* uses the function:

$$f(n) = g(n) + h(n)$$

- $g(n)$: cost from the start node to the current node.
- $h(n)$: estimated cost from the current node to the goal.
- This balance allows A* to avoid both greedy pitfalls and exhaustive searches.

5. Space Complexity

- A practical drawback: A* stores all generated nodes in memory.
- Worst-case space complexity is **exponential**, $O(b^d)$, where $b$is the branching factor and $d$is the depth of the solution.
- This makes A* memory-intensive for large problems.

6. Heuristic Dependence

- Performance depends heavily on the heuristic:
  - **Admissible heuristic** → guarantees optimality.
  - **Consistent heuristic** (monotonic) → ensures no node is revisited, improving efficiency.

Advantages

- Higher efficiency than BFS
- Uses Heuristic and path cost to find optimal cost while BFS searches blindly as it has no heuristic
- With a strong heuristic, it has better time complexity

Disadvantages

- Uses high memory
- Computing heuristics can be expensive slowing down the algorithm

Conclusion

In practice, A* with Manhattan distance solves the 8-puzzle efficiently and optimally while other algorithms (like IDA* or RBFS) are alternatives, but A* remains the most widely taught and applied because of its clarity and effectiveness.

- **Hill Climbing Algorithm for the 8-Puzzle Problem**
- **Algorithm Overview**

  Hill Climbing is a heuristic-based local search algorithm. It works by continuously moving from the current state to the neighboring state with the best heuristic value, aiming to reach the goal state.

- **Why Use Hill Climbing in the 8-Puzzle?**

  The Hill Climbing algorithm is suitable for the 8-Puzzle problem because it requires very low memory usage and provides fast performance for large state spaces.

- **Heuristic Function Used**

  • Number of Misplaced Tiles
  • Manhattan Distance

- **How Hill Climbing Solves the 8-Puzzle?**

  Step 1: Represent the Initial State
  The puzzle is represented as a 3×3 grid containing numbers from 1 to 8 and one empty tile (0).
  Step 2: Define the Goal State
  The goal state is predefined.
  Step 3: Calculate Heuristic Value
  The heuristic value of the current state is calculated.
  Step 4: Find the Empty Tile
  The position of the empty tile is identified.
  Step 5: Generate Neighbor States
  All valid moves (UP, DOWN, LEFT, RIGHT) are generated.
  Step 6: Evaluate Neighbor States
  Heuristic values are calculated for each neighboring state.
  Step 7: Choose the Best Neighbor
  The neighbor with the lowest heuristic value is selected.

Step 8: Goal Test
If the heuristic value becomes zero, the goal is reached.
Step 9: Termination
The algorithm terminates when no better neighbor exists.

- **Properties of Hill Climbing**

  Completeness:
  Not guaranteed.

- **Optimality:**
  Not guaranteed.

- **Time Complexity:**
  Depends on the heuristic function and branching factor.
  Space Complexity:
  O(1)

- **Advantages**
  • Simple and fast
  • Low memory usage

  Disadvantages
  • Can get stuck in local maxima
  • No guarantee of optimal solution

  Conclusion
  Hill Climbing is a fast and memory-efficient approach for solving the 8-Puzzle problem, but it may fail to find a solution in some cases.

➔ Comparison of Search Algorithms for the 8-Puzzle Problem

- In this project, several uninformed and informed search algorithms were implemented to solve the 8-Puzzle Problem. Each algorithm has different characteristics in terms of optimality, completeness, time complexity, and memory usage.

1. Breadth-First Search (BFS)

BFS explores the state space level by level and guarantees finding the shortest solution path if one exists. It is complete and optimal because all moves have equal cost.
However, BFS suffers from very high memory consumption, as it stores all generated states at each depth. For complex or deep puzzle configurations, BFS becomes impractical due to memory limitations.

Best use: When the state space is small and memory is not a concern.

---

2. Depth-First Search (DFS)

DFS explores one branch deeply before backtracking. It uses much less memory than BFS, making it space-efficient.
However, DFS is not optimal and not complete without a depth limit. It may get stuck exploring deep, unpromising paths and can return a non-optimal solution.

Best use: Demonstration purposes or problems with very small depth.

---

3. Uniform Cost Search (UCS)

UCS expands the node with the lowest cumulative cost, guaranteeing the optimal solution when all step costs are non-negative.
In the 8-Puzzle, where each move has cost 1, UCS behaves similarly to BFS but with a priority queue. Like BFS, it is complete and optimal, but it also suffers from high memory usage and can be slow.

Best use: Problems where action costs are different and optimality is required.

4. Iterative Deepening Search (IDS)

IDS combines the advantages of BFS and DFS. It performs repeated depth-limited DFS while gradually increasing the depth limit.
IDS is complete and optimal, like BFS, but uses significantly less memory, similar to DFS. The main drawback is repeated node expansion, which increases time overhead.

Best use: When solution depth is unknown and memory is limited.

---

5. A* Search (with Manhattan Distance)

A* is an informed search algorithm that uses both:

- Actual cost from the start (g(n))
- Estimated cost to the goal using a heuristic (h(n))

Using the Manhattan Distance heuristic, A* efficiently guides the search toward the goal while still guaranteeing optimality and completeness, because the heuristic is admissible.
A* expands far fewer nodes compared to BFS, UCS, and IDS, making it much faster in practice. Its main disadvantage is high memory usage, as it stores many generated states.

Best use: Finding optimal solutions efficiently in large state spaces.

---

6. Hill Climbing

Hill Climbing is a local search algorithm that only looks at the current state and its neighbors. It is very fast and memory-efficient, but it is neither complete nor optimal.
The algorithm can easily get stuck in local maxima or plateaus, meaning it may fail to find a solution even when one exists.

Best use: Fast approximations when optimality is not required.

➔ **Final Conclusion: Which Algorithm Is the Best?**

- A* Search with the Manhattan Distance heuristic is the best algorithm for solving the 8-Puzzle Problem.

➔ **Why A* is the best:**

- Complete (always finds a solution if one exists)
- Optimal (finds the shortest solution)
- Much faster than uninformed searches
- Uses heuristic knowledge to reduce unnecessary exploration

- Although A* requires more memory, its efficiency and optimality make it the most practical and effective algorithm among all those implemented.

Overall ranking (best to worst for 8-Puzzle):

1. **A***
2. Iterative Deepening Search (IDS)
3. Breadth-First Search (BFS)
4. Uniform Cost Search (UCS)
5. Depth-First Search (DFS)
6. Hill Climbing