**Date: 13th January 2025**

**Experiment 2**

**AIM:** Solve The-8 Puzzle problem.

**Introduction:** The 8-puzzle is a classic sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one blank space. The objective is to rearrange the tiles from a given initial state to a specified goal state by sliding the tiles into the blank space. Only horizontal or vertical moves are allowed.

**Initial State:** The starting configuration of the tiles on the grid.

**Goal State:** The desired final configuration of the tiles.

A common goal configuration is:

1 2 3

8 0 4

7 6 5

where 0 represents the blank space.

**Rules:**

1. Only tiles adjacent to the blank space can be moved into it.
2. A valid move is shifting one tile **up**, **down**, **left**, or **right** into the blank space.

**Solving techniques for the 8-Puzzle:**

Various search algorithms can be used to solve the 8-puzzle, including:

1. **Breadth-First Search (BFS):**

* Expands nodes in layers, ensuring the shortest path is found.
* Can be memory-intensive for large search spaces.

1. **Depth-First Search (DFS):**

* Explores one branch of the search tree as deeply as possible before backtracking.
* May not find the optimal solution and can get stuck in infinite loops.

1. *A*\*

* A heuristic search algorithm that uses a cost function (f(n) = g(n) + h(n)) to guide the search.
* g(n) is the cost to reach the current node.
* h(n) is an estimate of the cost to reach the goal from the current node (heuristic function).
* A\* search is often more efficient than BFS and DFS for solving the 8-puzzle.

**Applications:**

The 8-puzzle problem is used to study:

1. **Learn AI concepts:** It introduces heuristic search techniques in an accessible and educational way.
2. **Understand state-space search:** Demonstrates state-space exploration and optimization and how to explore possible solutions.
3. **Test search algorithms:** Like BFS, DFS, and A\*.

Code:

|  |
| --- |
| **BFS:** |
| **//Including header files and Namespace**  #include <bits/stdc++.h>  using namespace std;  **//Initialising the goal state**  const vector<vector<int>> goal = {      {1, 2, 3},      {8, 0, 4},      {7, 6, 5}  };  **//Function to print the puzzle**  void printPuzzle(const vector<vector<int>>& state) {      for (int i = 0; i < 3; i++) {          for (int j = 0; j < 3; j++)              cout << state[i][j] << " ";          cout << endl;      }      cout << "-------------\n";  }  **//Function to provide co-ordinates of '0'**  pair<int, int> findBlank(const vector<vector<int>>& state) {      for (int i = 0; i < 3; i++) {          for (int j = 0; j < 3; ++j) {              if (state[i][j] == 0)                  return {i, j};          }      }      return {-1, -1}; // Impossible  }  **//Function to check if the state is the goal state**  bool isGoal(const vector<vector<int>>& state) {      return state == goal;  }  **// Generate all possible moves from the current state**  vector<vector<vector<int>>> getNeighbors(const vector<vector<int>>& state) {        vector<vector<vector<int>>> neighbors;      pair<int, int> blank = findBlank(state);      int x = blank.first;      int y = blank.second;  **// Set possible moves (up, down, left, right)**      vector<pair<int, int>> moves = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};      for (const auto& move : moves) {          int nx = x + move.first;          int ny = y + move.second;  **// Check if the move is safe**          if (nx >= 0 && nx < 3 && ny >= 0 && ny < 3) {              vector<vector<int>> newState = state;              swap(newState[x][y], newState[nx][ny]);              neighbors.push\_back(newState);          }      }      return neighbors;  }  **// BFS implementation**  void solvePuzzle(const vector<vector<int>>& initial) {      queue<vector<vector<int>>> q;      vector<vector<vector<int>>> visited;      vector<pair<vector<vector<int>>, vector<vector<int>>>> parent; // To track parent-child relationships  **// Start BFS**      q.push(initial);      visited.push\_back(initial);      parent.push\_back({initial, {}}); **// Root has no parent**      while (!q.empty()) {          vector<vector<int>> current = q.front();          q.pop();  **// Check if goal is reached**          if (isGoal(current)) {              cout << "Goal state reached!" << endl;  **// Count the number of moves**              int steps = 0;              while (true) {                  auto it = find\_if(parent.begin(), parent.end(), [&](const auto& p) {                      return p.first == current;                  });                  if (it == parent.end() || it->second.empty())                      break;                  current = it->second;                  steps++;              }              printPuzzle(goal);              cout << "\nNo. of moves: " << steps << endl;              return;          }  **// Explore neighbours**          for (const auto& neighbor : getNeighbors(current)) {              if (find(visited.begin(), visited.end(), neighbor) == visited.end()) {                  q.push(neighbor);                  visited.push\_back(neighbor);                  parent.push\_back({neighbor, current}); **// Track the parent**              }          }      }      cout << "No solution found for this initial state!" << endl;  }  int main() {      vector<vector<int>> initial(3, vector<int>(3));      cout << "Enter the initial state matrix:" << endl;      for (int i = 0; i < 3; ++i) {          for (int j = 0; j < 3; ++j) {              cin >> initial[i][j];          }      }      cout << "\nInitial State:" << endl;      printPuzzle(initial);      cout << "\nWork in progess...\n" << endl;      cout << "-------------\n";      solvePuzzle(initial);      return 0;  } |
| **Output:** |

**Approach and Justification:  
-> Breadth-First Search (BFS)**

* **Description:** BFS explores all possible states level by level, ensuring that the shortest solution (least moves) is found first.
* **Shortest Path Guarantee:** BFS explores the search space level by level. This ensures that the first solution found is guaranteed to be the shortest path to the goal state.
* **Completeness:** BFS is complete, meaning it will always find a solution if one exists.
* **Memory Efficiency:** While BFS can use more memory than DFS in some cases, it's generally more memory-efficient for the 8-puzzle due to the relatively small search space.
* **Disadvantages:**
  + High memory consumption because it stores all states at the current level in the queue and exponential growth of state space makes it infeasible for complex puzzles.
* **Use Case:** BFS is ideal for small puzzles or when the shortest path to the solution is critical.

**Why DFS Might Not Be Ideal:**

* **May Not Find Shortest Path:** DFS explores deeply along one branch before backtracking. This can lead to finding longer solutions or getting stuck in infinite loops.

BFS is generally preferred for the 8-puzzle because it provides the shortest solution and is relatively efficient in terms of both time and space complexity for this problem.

**However, it's important to note:**

*A*\*: For more complex problems or larger state spaces, A\* search, which uses a heuristic function to guide the search, can be significantly more efficient than both BFS and DFS.