**Date: 20th January 2025**

**Experiment 2**

**AIM:** Solve The-8 Puzzle problem using A\* algorithm where initial state and goal state will be given by the user.

**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.

**Search Algorithm Used**

The program implements the **A\* (A-star) search algorithm**, which is a heuristic search algorithm designed to find the shortest path from the initial state to the goal state.

* This algorithm 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).
* For the heuristic function, the **Manhattan Distance** is used, which calculates the total number of moves each tile needs to reach its correct position in the goal state.
* 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.

* **Real world applications:** In areas like robot navigation, logistics, and pathfinding.

**Code:**

|  |
| --- |
| ***A\**** |
| #include <bits/stdc++.h>  using namespace std;  **// Possible moves (up, down, left, right) for 0**  const vector<vector<int>> DIRECTIONS = { {-1, 0}, {1, 0}, {0, -1}, {0, 1} };  const string GOAL = "123804765";  **// Count inversions in a 1D vector (ignoring 0)**  int countInversions(const vector<int>& puzzle) {      int invCount = 0;      int n = puzzle.size();      for (int i = 0; i < n; i++) {          if (puzzle[i] == 0) continue;          for (int j = i + 1; j < n; j++) {              if (puzzle[j] != 0 && puzzle[i] > puzzle[j])                  invCount++;          }      }      return invCount;  }  **// To check if a given puzzle state is solvable**  bool isSolvable(const vector<int>& initPuzzle, const vector<int>& goalPuzzle) {      int initInv = countInversions(initPuzzle);      int goalInv = countInversions(goalPuzzle);  **// Puzzle is solvable if inversion parities match**      return (initInv % 2 == goalInv % 2);  }  **// Heuristic: Manhattan distance**  int heuristic(const string& state) {      int distance = 0;      vector<pair<int, int>> goalPos(9);  **// Converting string into matrix**      for (int i = 0; i < 9; i++) {          int num = GOAL[i] - '0';          goalPos[num] = make\_pair(i / 3, i % 3);      }      for (int i = 0; i < 9; i++) {          int num = state[i] - '0';          if (num != 0)  **// Calculating distance if number is not 0**              distance += abs(goalPos[num].first - i / 3) + abs(goalPos[num].second - i % 3);      }      return distance;  }  **// To Check if the move is within the board and is not illegal**  bool isValid(int x, int y) {      return x >= 0 && x < 3 && y >= 0 && y < 3;  }  **// Conversion into string**  string matrixToString(const vector<vector<int>>& matrix) {      string result;      for (int i = 0; i < 3; i++)          for (int j = 0; j < 3; j++)              result += to\_string(matrix[i][j]);      return result;  }  **// To swap numbers in string format itself**  string swapTiles(const string& state, int x1, int y1, int x2, int y2) {      string nextState = state;      int idx1 = x1 \* 3 + y1;      int idx2 = x2 \* 3 + y2;      swap(nextState[idx1], nextState[idx2]);      return nextState;  }  **// To print state**  void printState(const string& state) {      for (int i = 0; i < (int)state.size(); i++) {          if (i > 0 && i % 3 == 0) cout << endl;          cout << state[i] << " ";      }      cout << "\n------";      cout << endl;  }  **// To print the reconstructed path**  void printSolution(int goalIndex, const vector<string>& states, const vector<int>& parent) {      vector<string> path;      int current = goalIndex;      while (current != -1) {          path.push\_back(states[current]);          current = parent[current];      }      reverse(path.begin(), path.end());      for (const auto &s : path) {          printState(s);          cout << endl;      }      cout << "Number of steps required: " << path.size() - 1 << endl;  }  **// Implementation of A\***  void solve(const vector<vector<int>>& start) {    **// Convert start matrix into a vector for inversion counting.**      vector<int> initPuzzle;      for (const auto &row : start)          for (int num : row)              initPuzzle.push\_back(num);    **// Convert goal state string into a vector.**      vector<int> goalPuzzle;      for (char ch : GOAL)          goalPuzzle.push\_back(ch - '0');  **// Check solvability**      if (!isSolvable(initPuzzle, goalPuzzle)) {          cout << "The given puzzle is unsolvable." << endl;          return;      }    **// Representation of initial state**      string startState = matrixToString(start);    **// (f = g + heuristic, g, state index)**      typedef tuple<int, int, int> Node;  **// To order nodes by the smallest 'f' value using greater<Node>**      priority\_queue<Node, vector<Node>, greater<Node>> pq;  **// To map state string to its respective State index**      unordered\_map<string, int> stateIndex;  **// To store all the state strings ever encountered**      vector<string> states;  **// Cost to reach each state**      vector<int> g\_values;    **// To store parent index of the state**      vector<int> parent;  **// Entering the start state in the priority queue**      states.push\_back(startState);      g\_values.push\_back(0);      parent.push\_back(-1);      stateIndex[startState] = 0;      pq.push(make\_tuple(heuristic(startState), 0, 0));    **// Closed set to avoid re-exploring states.**      unordered\_set<string> closed;  **// A\* Loop**      while (!pq.empty()) {          int f, g, currentIdx;  **// Node with the smallest 'f' is popped**          tie(f, g, currentIdx) = pq.top();          pq.pop();    **// Checking if the state is in the closed list**          string currentState = states[currentIdx];          if (closed.find(currentState) != closed.end())              continue;          closed.insert(currentState);    **// Checking for the current state to be the goal state**          if (currentState == GOAL) {              printSolution(currentIdx, states, parent);              cout << "Goal state reached Successfully!" << endl;              return;          }    **// Finding 0 in the current state string**          int zeroIndex = currentState.find('0');  **// Getting its co-ordinates**          int x = zeroIndex / 3, y = zeroIndex % 3;    **// Move in possible 4 directions and check and update f,g,h**          for (int i = 0; i < 4; i++) {              int newX = x + DIRECTIONS[i][0];              int newY = y + DIRECTIONS[i][1];                if (isValid(newX, newY)) {                  string nextState = swapTiles(currentState, x, y, newX, newY);                  int newG = g + 1;                  int newF = newG + heuristic(nextState);    **// New state found then add it in the list and queue**                  if (stateIndex.find(nextState) == stateIndex.end()) {                      int newIdx = states.size();                      states.push\_back(nextState);                      g\_values.push\_back(newG);                      parent.push\_back(currentIdx);                      stateIndex[nextState] = newIdx;                      pq.push(make\_tuple(newF, newG, newIdx));                  } else {  **// Old state but lower cost so reconsidering**                      int existingIdx = stateIndex[nextState];                      if (newG < g\_values[existingIdx]) {                          g\_values[existingIdx] = newG;                          parent[existingIdx] = currentIdx;                          pq.push(make\_tuple(newF, newG, existingIdx));                      }                  }              }          }      }      cout << "No solution found:(" << endl;  }  int main() {      vector<vector<int>> start = {          {8, 3, 5},          {4, 1, 6},          {2, 7, 0}      };  **// Printing start state**      cout << "Initial state:" << endl;      for (int i = 0; i < 3; i++) {          for (int j = 0; j < 3; j++) {              cout << start[i][j];              cout << " ";          }          cout << "\n";      }      cout << endl;      solve(start);      return 0;  } |
| **Output:** |

**Comparison:**

|  |  |
| --- | --- |
| Using BFS | Using A\* |
| 1. More steps (explores blindly). Explored total of around 40,000 states. 2. Stores more states 3. Slower due to exhaustive search | 1. Fewer steps (more efficient path). Explored total of 2000 states. 2. Stores fewer states 3. Faster due to heuristic |

**Why is A\* Better?**

1. **Heuristic Helps**: A\* uses **Manhattan distance** to prioritize promising paths.
2. **Memory Efficient**: A\* avoids storing unnecessary states by selecting optimal paths.
3. **Faster Execution**: By selecting better moves, A\* reduces unnecessary state expansions.