-------------------------------------------------

8-Puzzle Heuristic Solver: Line-by-Line Explanation

-------------------------------------------------

-------------------------------------------------

[1] g = 0

- Global variable to keep track of the number of moves (g-cost).

-------------------------------------------------

[2-8] def print\_board(elements):

for i in range(9):

if i % 3 == 0:

print()

if elements[i] == -1:

print("\_", end=" ")

else:

print(elements[i], end=" ")

print()

- Function to print the 3x3 board.

- Displays “\_” for the empty tile (-1).

-------------------------------------------------

[10-20] def solvable(start):

inv = 0

for i in range(9):

if start[i] <= 1:

continue

for j in range(i + 1, 9):

if start[j] == -1:

continue

if start[i] > start[j]:

inv += 1

if inv % 2 == 0:

return True

return False

- Checks if the puzzle configuration is solvable.

- Uses the concept of \*\*inversions\*\*.

- If the number of inversions is even, it’s solvable.

-------------------------------------------------

[22-29] def heuristic(start, goal):

global g

h = 0

for i in range(9):

for j in range(9):

if start[i] == goal[j] and start[i] != -1:

h += (abs(j - i)) // 3 + (abs(j - i)) % 3

return h + g

- Heuristic function.

- Measures the Manhattan distance of tiles from their goal positions.

- Returns f = g + h.

-------------------------------------------------

[31-49] Movement Functions:

- moveleft(), moveright(), moveup(), movedown()

- Swap the empty tile (-1) with the neighbor in the specified direction.

-------------------------------------------------

[52-77] def movetile(start, goal):

emptyat = start.index(-1)

row = emptyat // 3

col = emptyat % 3

t1, t2, t3, t4 = start[:], start[:], start[:], start[:]

f1, f2, f3, f4 = 100, 100, 100, 100

if col - 1 >= 0:

moveleft(t1, emptyat)

f1 = heuristic(t1, goal)

if col + 1 < 3:

moveright(t2, emptyat)

f2 = heuristic(t2, goal)

if row + 1 < 3:

movedown(t3, emptyat)

f3 = heuristic(t3, goal)

if row - 1 >= 0:

moveup(t4, emptyat)

f4 = heuristic(t4, goal)

min\_heuristic = min(f1, f2, f3, f4)

if f1 == min\_heuristic:

moveleft(start, emptyat)

elif f2 == min\_heuristic:

moveright(start, emptyat)

elif f3 == min\_heuristic:

movedown(start, emptyat)

elif f4 == min\_heuristic:

moveup(start, emptyat)

- Checks all four possible moves and chooses the one with the minimum heuristic.

-------------------------------------------------

[80-89] def solveEight(start, goal):

global g

g += 1

movetile(start, goal)

print\_board(start)

f = heuristic(start, goal)

if f == g:

print("Solved in {} moves".format(f))

return

solveEight(start, goal)

- Recursive solver.

- Moves step by step until the heuristic f equals the number of moves.

-------------------------------------------------

[92-108] def main():

global g

start = list()

goal = list()

print("Enter the start state:(Enter -1 for empty):")

for i in range(9):

start.append(int(input()))

print("Enter the goal state:(Enter -1 for empty):")

for i in range(9):

goal.append(int(input()))

print\_board(start)

if solvable(start):

solveEight(start, goal)

print("Solved in {} moves".format(g))

else:

print("Not possible to solve")

- Main function to take input, check solvability, and start solving.

-------------------------------------------------

[111-112] if \_\_name\_\_ == '\_\_main\_\_':

print("Lokesh Dhoble TACO22111")

main()

- Entry point of the program.

- Runs the main() function with your name and roll.

-------------------------------------------------

Possible Viva Questions

-------------------------------------------------

1️⃣ What is the 8-puzzle problem?

- Rearrange tiles on a 3x3 board to reach the goal configuration.

2️⃣ What search strategy is used here?

- Greedy heuristic-based search (best move at each step).

3️⃣ What heuristic is used?

- Manhattan distance: how far each tile is from its goal.

4️⃣ How do you check if a puzzle is solvable?

- Count the number of inversions; solvable if even.

5️⃣ Why use -1 for the empty tile?

- Makes it easy to identify the blank space programmatically.

6️⃣ Why do you create copies (t1, t2, t3, t4)?

- To simulate all possible moves without changing the original state.

7️⃣ What’s the role of the global variable g?

- Tracks the number of moves made so far.

8️⃣ Does this method guarantee the shortest solution?

- No; it’s greedy, not guaranteed to find the optimal solution.

9️⃣ What is a more optimal algorithm?

- A\* with proper open/closed lists, or IDA\*.

🔟 Applications of 8-puzzle solver?

- AI search techniques, game solvers, robotic pathfinding, puzzle games.

-------------------------------------------------

Let me know if you want this written out into a `.txt` file for download!

**Detailed Line-by-Line Explanation: 8-Puzzle Solver Code**

g = 0

**Global variable to count moves taken.**

def print\_board(elements):  
for i in range(9):  
if i % 3 == 0:  
print()  
if elements[i] == -1:  
print("\_", end=" ")  
else:  
print(elements[i], end=" ")  
print()

**Prints the board as a 3x3 grid, showing '\_' for empty (-1).**

def solvable(start):  
inv = 0  
for i in range(9):  
if start[i] <= 1:  
continue  
for j in range(i + 1, 9):  
if start[j] == -1:  
continue  
if start[i] > start[j]:  
inv += 1  
if inv % 2 == 0:  
return True  
return False

**Counts inversions to check if the puzzle can be solved.**

def heuristic(start, goal):  
global g  
h = 0  
for i in range(9):  
for j in range(9):  
if start[i] == goal[j] and start[i] != -1:  
h += (abs(j - i)) // 3 + (abs(j - i)) % 3  
return h + g

**Heuristic function: Manhattan distance + moves taken (g).**

def moveleft(start, position):  
start[position], start[position - 1] = start[position - 1], start[position]

**Swaps the empty tile with the left neighbor.**

def moveright(start, position):  
start[position], start[position + 1] = start[position + 1], start[position]

**Swaps the empty tile with the right neighbor.**

def moveup(start, position):  
start[position], start[position - 3] = start[position - 3], start[position]

**Swaps the empty tile with the upper neighbor.**

def movedown(start, position):  
start[position], start[position + 3] = start[position + 3], start[position]

**Swaps the empty tile with the lower neighbor.**

def movetile(start, goal):  
emptyat = start.index(-1)  
row = emptyat // 3  
col = emptyat % 3  
t1, t2, t3, t4 = start[:], start[:], start[:], start[:]  
f1, f2, f3, f4 = 100, 100, 100, 100

**Prepare copies for each possible move and set high initial heuristic values.**

if col - 1 >= 0:

moveleft(t1, emptyat)

f1 = heuristic(t1, goal)

if col + 1 < 3:

moveright(t2, emptyat)

f2 = heuristic(t2, goal)

if row + 1 < 3:

movedown(t3, emptyat)

f3 = heuristic(t3, goal)

if row - 1 >= 0:

moveup(t4, emptyat)

f4 = heuristic(t4, goal)

**Check all valid moves and calculate heuristic for each.**

min\_heuristic = min(f1, f2, f3, f4)

**Choose the move with the lowest heuristic value.**

if f1 == min\_heuristic:

moveleft(start, emptyat)

elif f2 == min\_heuristic:

moveright(start, emptyat)

elif f3 == min\_heuristic:

movedown(start, emptyat)

elif f4 == min\_heuristic:

moveup(start, emptyat)

**Apply the best move.**

def solveEight(start, goal):  
global g  
g += 1  
movetile(start, goal)  
print\_board(start)  
f = heuristic(start, goal)  
if f == g:  
print("Solved in {} moves".format(f))  
return  
solveEight(start, goal)

**Recursive function that keeps applying the best move until solved.**

def main():  
global g  
start = []  
goal = []  
print("Enter the start state:(Enter -1 for empty):")  
for i in range(9):  
start.append(int(input()))

print("Enter the goal state:(Enter -1 for empty):")

for i in range(9):

goal.append(int(input()))

print\_board(start)

if solvable(start):

solveEight(start, goal)

print("Solved in {} moves".format(g))

else:

print("Not possible to solve")

**Gets input, checks solvability, starts solving.**

if **name** == '**main**':  
print("Lokesh Dhoble TACO22111")  
main()

**Program entry point.**