Assignment Questions:

1. Depth-First Search (DFS) Algorithm:
    a. Explain how the DFS algorithm works, and describe it using pseudocode or an algorithm.

DFS algorithm works using a searching method which goes to the depth of a node. When a dead end is reached, the search backs up to the next shallowest node that still has unexplored successors. This strategy can be implemented by tree search using a last-in-first-out (LIFO), also known as a stack. It always expands the deepest node in the current fringe of the search tree.

DFS Algorithm:

1. Initialize an empty stack S

2. Initialize an empty set Visited

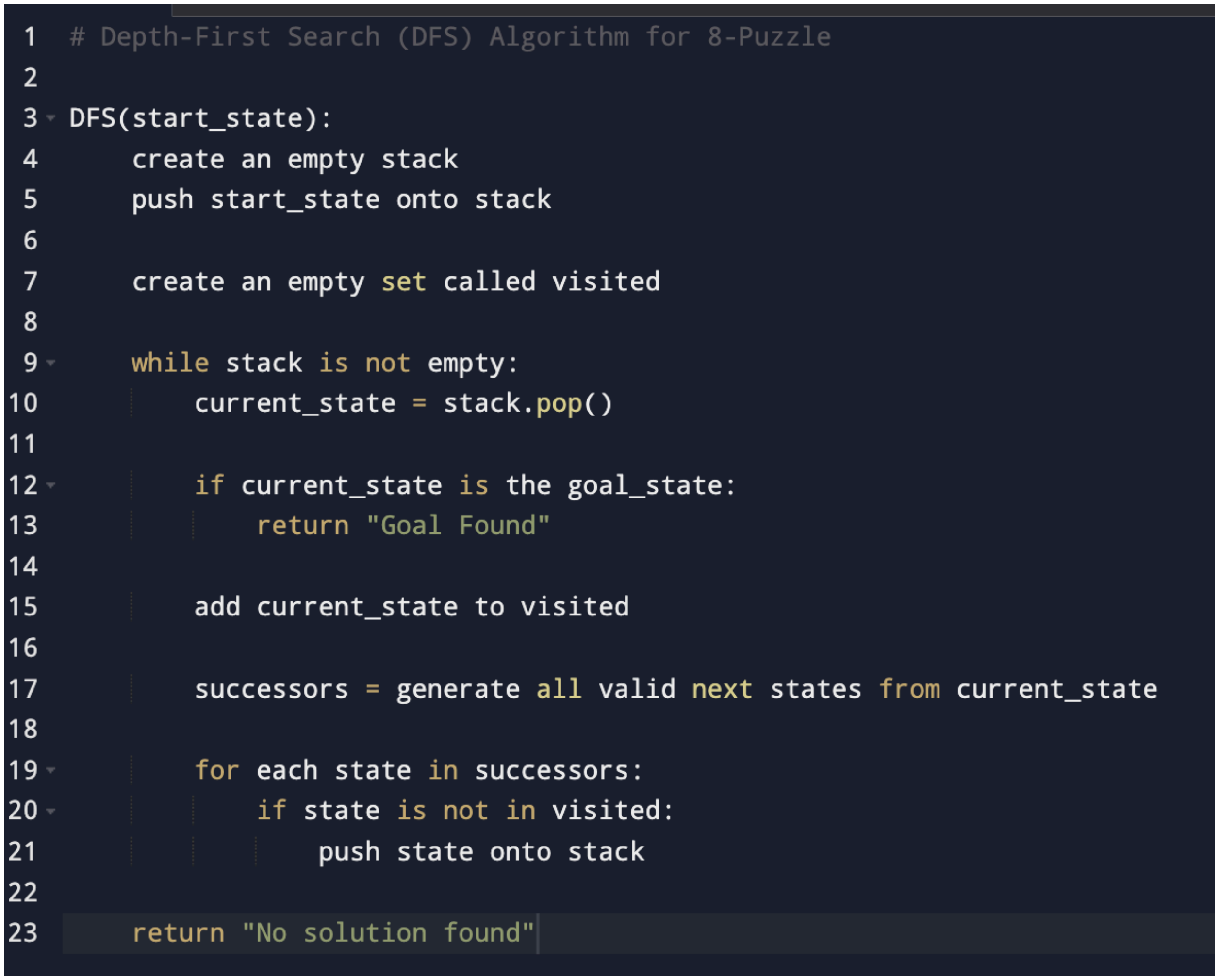
3. Push start onto S

4. Add start to Visited

5. While S is not empty, do:

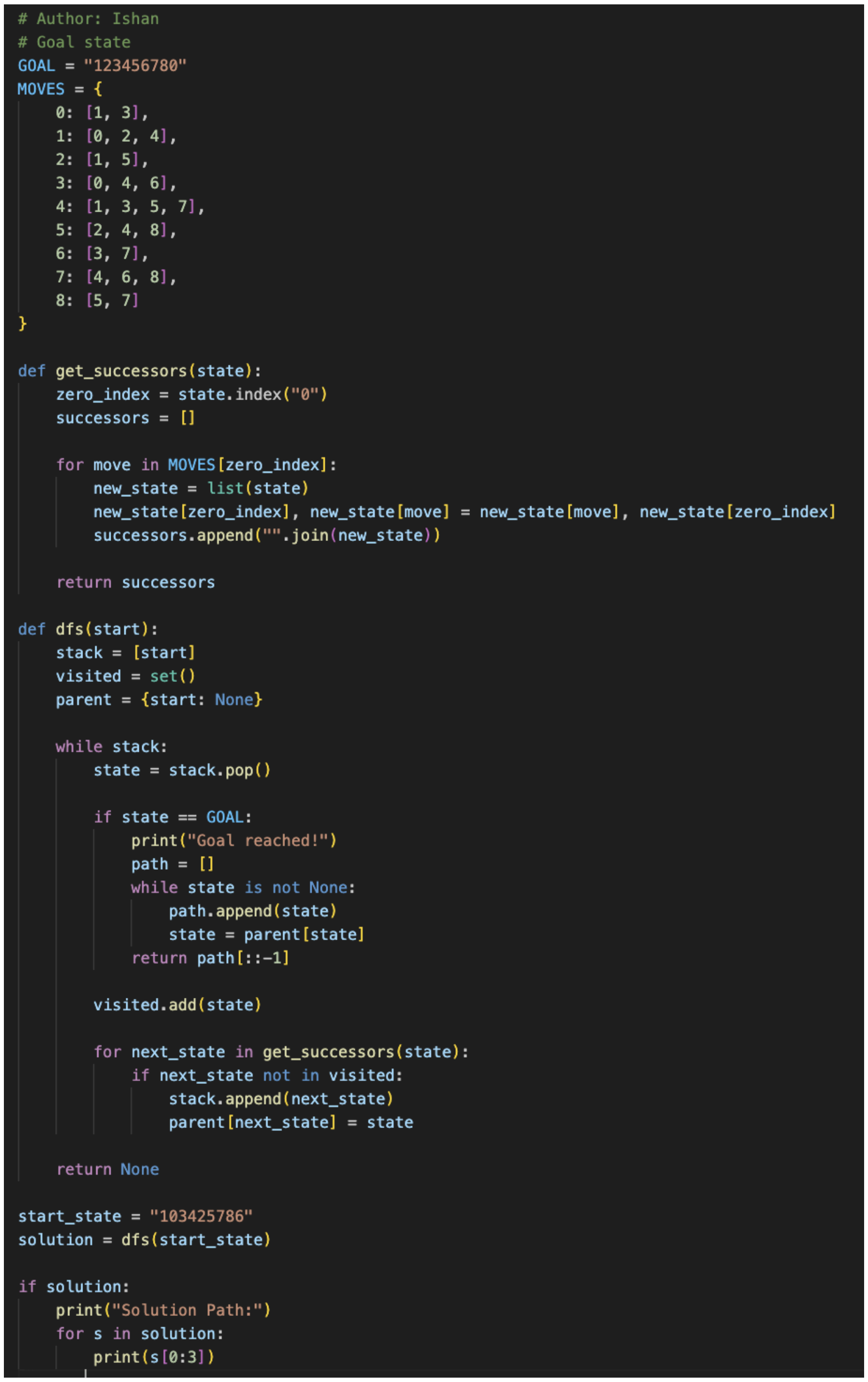
- Pop a node current from S
- Process current (e.g., print or store it)
- For each neighbor n of current in G (in reverse order if needed for consistent ordering):
- If n is not in Visited:
- Push n onto S
- Add n to Visited

b. Implement the 8-puzzle problem using the DFS algorithm in Python.

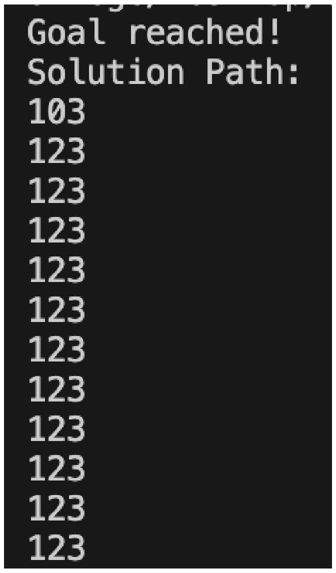


```
# Depth-First Search (DFS) Algorithm for 8-Puzzle

DFS(start_state):
    create an empty stack
    push start_state onto stack

    create an empty set called visited

    while stack is not empty:
        current_state = stack.pop()

        if current_state is the goal_state:
            return "Goal Found"

        add current_state to visited

        successors = generate all valid next states from current_state

        for each state in successors:
            if state is not in visited:
                push state onto stack

    return "No solution found"
```

c. Include screenshots of your code and its output. Please write your name as a comment in the code.



```python
# Author: Ishan
# Goal state
GOAL = "123456780"
MOVES = {
    0: [1, 3],
    1: [0, 2, 4],
    2: [1, 5],
    3: [0, 4, 6],
    4: [1, 3, 5, 7],
    5: [2, 4, 8],
    6: [3, 7],
    7: [4, 6, 8],
    8: [5, 7]
}

def get_successors(state):
    zero_index = state.index("0")
    successors = []

    for move in MOVES[zero_index]:
        new_state = list(state)
        new_state[zero_index], new_state[move] = new_state[move], new_state[zero_index]
        successors.append("".join(new_state))

    return successors

def dfs(start):
    stack = [start]
    visited = set()
    parent = {start: None}

    while stack:
        state = stack.pop()

        if state == GOAL:
            print("Goal reached!")
            path = []
            while state is not None:
                path.append(state)
                state = parent[state]
            return path[::-1]

        visited.add(state)

        for next_state in get_successors(state):
            if next_state not in visited:
                stack.append(next_state)
                parent[next_state] = state

    return None

start_state = "103425786"
solution = dfs(start_state)

if solution:
    print("Solution Path:")
    for s in solution:
        print(s[0:3])
```



```
Goal reached!
Solution Path:
103
123
123
123
123
123
123
123
123
123
123
123
```

d. Discuss why, in some cases, DFS may fail to find a solution for this problem.

- The 8-puzzle contains repeated states. Without cycle detection, DFS loops forever.
- DFS may go down a long wrong path of depth thousands before reaching an actual solution.
- DFS doesn't guarantee finding the shortest solution.
- If the depth limit is small, the solution may lie deeper.

2. Breadth-First Search (BFS) State Space Design:
    a. While implementing BFS in Python, did you first construct the entire state space (tree or graph) before starting the search, or did you build it dynamically during the search?

While implementing BFS in Python, we first build it dynamically during the search. Dynamic construction builds only the states you actually reach. BFS naturally builds level-by-level, so it fits dynamic expansion. In this way, it reduces both time and speed and is the optimal way to search using BFS.

b. Explain the approach you took, and justify why you chose that method.

I took the dynamic searching method. The reasons are given below:

- Dynamic construction builds only the states you actually reach.
- BFS naturally builds level-by-level, so it fits dynamic expansion.
- It reduces both time and speed.

3. Informed Search vs. BFS:
    a. What do you understand by the term informed search?

Informed search algorithms use problem-specific knowledge (heuristics) to guess which path is more likely to lead to the goal. They try to be "smart" about which paths to explore first.

b. How does informed search differ from Breadth-First Search (BFS)?

Feature	BFS	Informed Search
Heuristic	None	Uses heuristic
Search order	Level-by-level	Goal-directed
Speed	Slow	Much faster

c. Explain the differences with the help of an example.

Example:

Finding shortest path in a maze:

- BFS: explores in all directions
- A*: moves directly toward the goal using heuristic (distance)

4. Heuristic Values in Informed Search:

a. Define heuristic values in the context of informed search algorithms.

A heuristic is an estimate of how close a node is to the goal.

Example: A* moves from the start string to the goal string by always choosing the next state with the smallest value of $f = g + h$, where h counts how many bits differ from the goal.

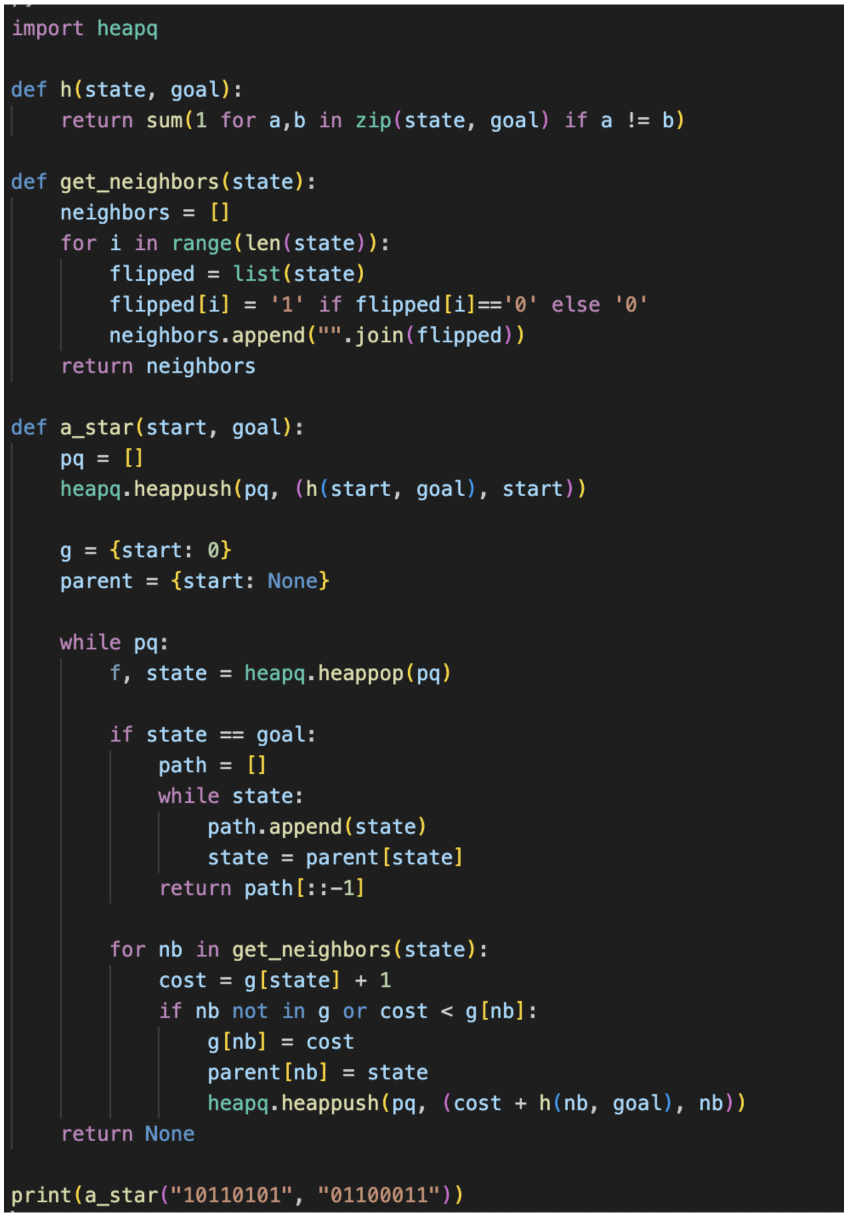
b. List a few common heuristic functions and mention their application areas.

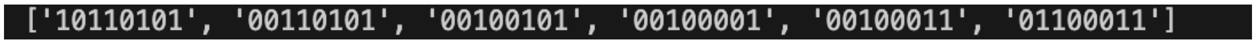
Heuristic	Used In	Application
Manhattan Distance	A*	Grid pathfinding, 8-puzzle
Hamming Distance	A*	String matching, 8-puzzle
Euclidean Distance	Gaming AI	Continuous movement
Misplaced Tiles	A*	8-puzzle

5. A* Algorithm for Binary String Matching:
   a. How can the A* algorithm be applied to solve the problem of binary string matching?
      Example: From the start state 10110101 to the goal state 01100011.

How A* applies to binary string matching:
Each state is a binary string.
Operators:
o Flip a bit
o Swap two bits
Cost g(n) = number of operations so far
Heuristic h(n): number of mismatched bits with goal (Hamming distance)

   b. Implement the A* algorithm in Python to solve this problem.
   c. Include screenshots of your code and output, and write your name as a comment in the code.


```python
import heapq

def h(state, goal):
    return sum(1 for a,b in zip(state, goal) if a != b)

def get_neighbors(state):
    neighbors = []
    for i in range(len(state)):
        flipped = list(state)
        flipped[i] = '1' if flipped[i]=='0' else '0'
        neighbors.append("".join(flipped))
    return neighbors

def a_star(start, goal):
    pq = []
    heapq.heappush(pq, (h(start, goal), start))

    g = {start: 0}
    parent = {start: None}

    while pq:
        f, state = heapq.heappop(pq)

        if state == goal:
            path = []
            while state:
                path.append(state)
                state = parent[state]
            return path[::-1]

        for nb in get_neighbors(state):
            cost = g[state] + 1
            if nb not in g or cost < g[nb]:
                g[nb] = cost
                parent[nb] = state
                heapq.heappush(pq, (cost + h(nb, goal), nb))
    return None

print(a_star("10110101", "01100011"))
```


```
['10110101', '00110101', '00100101', '00100001', '00100011', '01100011']
```

d. Which heuristic function did you use? Explain how your code computes the heuristic values.

Hamming distance heuristic function was used to count how many bits differ between current and goal string.

Example:

10110101

01100011

5 bits different → h = 5

This ensures A* always moves toward states with fewer mismatched bits.