The **8-puzzle** is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one blank space. The objective is to move the tiles to match a goal state by sliding one tile into the blank space at a time. Let's implement this using **BFS**, **DFS**, **UCS**, and **GBFS**.

**Problem Representation:**

* The board can be represented as a list or a tuple in Python, where the blank space is represented by 0.
* Moves: The blank tile can move up, down, left, or right.
* Goal state: A typical goal state is [1, 2, 3, 4, 5, 6, 7, 8, 0] where 0 represents the blank.

Let's first define helper functions for moving tiles and checking the goal.

**Helper Functions**

python

Copy code

# Function to find the index of the blank (0) space

def find\_blank(state):

return state.index(0)

# Function to swap tiles

def swap(state, i, j):

new\_state = list(state)

new\_state[i], new\_state[j] = new\_state[j], new\_state[i]

return tuple(new\_state)

# Function to generate possible moves from the current state

def get\_neighbors(state):

neighbors = []

blank = find\_blank(state)

# Define the possible moves (up, down, left, right)

row, col = divmod(blank, 3)

moves = []

if row > 0: # Up

moves.append(blank - 3)

if row < 2: # Down

moves.append(blank + 3)

if col > 0: # Left

moves.append(blank - 1)

if col < 2: # Right

moves.append(blank + 1)

# Generate the new states

for move in moves:

new\_state = swap(state, blank, move)

neighbors.append(new\_state)

return neighbors

# Function to check if the current state is the goal state

def is\_goal(state, goal):

return state == goal

**1. BFS (Breadth-First Search) for 8-Puzzle**

python

Copy code

from collections import deque

def bfs\_8\_puzzle(start, goal):

queue = deque([(start, [])]) # Queue stores (state, path)

visited = set()

visited.add(start)

while queue:

state, path = queue.popleft()

if is\_goal(state, goal):

return path

for neighbor in get\_neighbors(state):

if neighbor not in visited:

visited.add(neighbor)

queue.append((neighbor, path + [neighbor]))

# Example usage:

start = (1, 2, 3, 4, 5, 6, 7, 8, 0) # Initial state

goal = (1, 2, 3, 4, 5, 6, 7, 0, 8) # Goal state

solution = bfs\_8\_puzzle(start, goal)

print("Solution using BFS:", solution)

**Explanation**:

* **BFS** explores all states at the current depth before going deeper.
* It uses a queue to explore neighboring states of the current board configuration.
* It returns the solution as a sequence of board states leading to the goal.

**2. DFS (Depth-First Search) for 8-Puzzle**

python

Copy code

def dfs\_8\_puzzle(start, goal):

stack = [(start, [])] # Stack stores (state, path)

visited = set()

visited.add(start)

while stack:

state, path = stack.pop()

if is\_goal(state, goal):

return path

for neighbor in get\_neighbors(state):

if neighbor not in visited:

visited.add(neighbor)

stack.append((neighbor, path + [neighbor]))

# Example usage:

solution = dfs\_8\_puzzle(start, goal)

print("Solution using DFS:", solution)

**Explanation**:

* **DFS** explores one path fully before backtracking and trying another path.
* It uses a stack to explore neighboring states in depth-first manner.
* The search may go deep down one path before exploring other paths.

**3. UCS (Uniform Cost Search) for 8-Puzzle**

In **UCS**, all moves have the same cost, so it behaves similarly to BFS. However, we explicitly keep track of the cost in UCS.

python

Copy code

def ucs\_8\_puzzle(start, goal):

queue = [(0, start, [])] # Priority queue (cost, state, path)

visited = set()

visited.add(start)

while queue:

queue.sort(key=lambda x: x[0]) # Sort by cost

cost, state, path = queue.pop(0)

if is\_goal(state, goal):

return path

for neighbor in get\_neighbors(state):

if neighbor not in visited:

visited.add(neighbor)

queue.append((cost + 1, neighbor, path + [neighbor]))

# Example usage:

solution = ucs\_8\_puzzle(start, goal)

print("Solution using UCS:", solution)

**Explanation**:

* **UCS** keeps track of the cost (number of moves) and explores the states with the least cost first.
* Since all moves in 8-puzzle have equal cost, UCS behaves similarly to BFS but with an explicit cost structure.

**4. GBFS (Greedy Best-First Search) for 8-Puzzle**

In **GBFS**, we use a heuristic to guide the search. One common heuristic is the **Manhattan distance** (sum of the distances of each tile from its goal position).

python

Copy code

# Heuristic: Manhattan distance

def manhattan\_distance(state, goal):

distance = 0

for i in range(1, 9): # We don't calculate for 0 (blank)

state\_pos = state.index(i)

goal\_pos = goal.index(i)

state\_row, state\_col = divmod(state\_pos, 3)

goal\_row, goal\_col = divmod(goal\_pos, 3)

distance += abs(state\_row - goal\_row) + abs(state\_col - goal\_col)

return distance

def gbfs\_8\_puzzle(start, goal):

queue = [(manhattan\_distance(start, goal), start, [])] # (heuristic, state, path)

visited = set()

visited.add(start)

while queue:

queue.sort(key=lambda x: x[0]) # Sort by heuristic value

h\_value, state, path = queue.pop(0)

if is\_goal(state, goal):

return path

for neighbor in get\_neighbors(state):

if neighbor not in visited:

visited.add(neighbor)

h\_value = manhattan\_distance(neighbor, goal)

queue.append((h\_value, neighbor, path + [neighbor]))

# Example usage:

solution = gbfs\_8\_puzzle(start, goal)

print("Solution using GBFS:", solution)

**Explanation**:

* **GBFS** uses a heuristic to guide the search, prioritizing states that are "closer" to the goal based on the heuristic.
* The **Manhattan distance** is used to estimate how far a state is from the goal.

**Summary:**

* **BFS**: Explores level by level and guarantees the shortest solution in terms of moves.
* **DFS**: Explores one path deeply before backtracking, but may not find the shortest solution.
* **UCS**: Behaves like BFS with an explicit cost for each move.
* **GBFS**: Uses a heuristic (like Manhattan distance) to guide the search toward the goal, but it doesn't always guarantee the shortest path.

These algorithms solve the 8-puzzle problem by searching through possible board configurations until the goal state is reached.