import heapq

from itertools import count

from typing import Callable, Dict, Iterable, List, Optional, Tuple

from functools import lru\_cache

State = Tuple[int, ...] # 9-length tuple; 0 represents the blank

GOAL: State = (1, 2, 3, 4, 5, 6, 7, 8, 0)

# Directions for blank movement: (dr, dc, move\_char)

DIRS = [(-1, 0, 'U'), (1, 0, 'D'), (0, -1, 'L'), (0, 1, 'R')]

def index\_to\_rc(i: int) -> Tuple[int, int]:

return divmod(i, 3)

def rc\_to\_index(r: int, c: int) -> int:

return r \* 3 + c

def is\_solvable(state: State) -> bool:

"""An 8-puzzle is solvable iff the inversion count (ignoring 0) is even."""

arr = [x for x in state if x != 0]

inv = 0

for i in range(len(arr)):

for j in range(i + 1, len(arr)):

if arr[i] > arr[j]:

inv += 1

return inv % 2 == 0

def neighbors(state: State) -> Iterable[Tuple[State, str]]:

"""Yield (neighbor\_state, move\_char) for each legal slide."""

i0 = state.index(0)

r0, c0 = index\_to\_rc(i0)

for dr, dc, mv in DIRS:

nr, nc = r0 + dr, c0 + dc

if 0 <= nr < 3 and 0 <= nc < 3:

ni = rc\_to\_index(nr, nc)

new\_state = list(state)

new\_state[i0], new\_state[ni] = new\_state[ni], new\_state[i0]

yield tuple(new\_state), mv

def reconstruct\_path(parents: Dict[State, State], end: State) -> List[State]:

path = [end]

while end in parents:

end = parents[end]

path.append(end)

path.reverse()

return path

def path\_to\_moves(path: List[State]) -> str:

"""Convert a path of states into a string of moves 'UDLR'."""

moves = []

for a, b in zip(path, path[1:]):

ia, ib = a.index(0), b.index(0)

ra, ca = index\_to\_rc(ia)

rb, cb = index\_to\_rc(ib)

if rb == ra - 1 and cb == ca:

moves.append('U')

elif rb == ra + 1 and cb == ca:

moves.append('D')

elif rb == ra and cb == ca - 1:

moves.append('L')

elif rb == ra and cb == ca + 1:

moves.append('R')

else:

moves.append('?')

return ''.join(moves)

def misplaced\_tiles(state: State, goal: State = GOAL) -> int:

"""Heuristic: number of tiles not in their goal position (ignore blank)."""

return sum(1 for i, v in enumerate(state) if v != 0 and v != goal[i])

@lru\_cache(maxsize=None)

def \_goal\_positions(goal: State) -> Tuple[Tuple[int, int], ...]:

"""Cache goal positions as a tuple indexed by tile value (0..8)."""

pos = [(0, 0)] \* 9

for i, v in enumerate(goal):

r, c = index\_to\_rc(i)

pos[v] = (r, c)

return tuple(pos)

def manhattan\_distance(state: State, goal: State = GOAL) -> int:

"""Heuristic: sum of Manhattan distances of each tile to its goal position."""

pos = \_goal\_positions(goal)

total = 0

for i, v in enumerate(state):

if v == 0:

continue

r, c = index\_to\_rc(i)

gr, gc = pos[v]

total += abs(r - gr) + abs(c - gc)

return total

def a\_star(

start: State,

heuristic: Callable[[State, State], int] = manhattan\_distance,

goal: State = GOAL,

) -> Optional[Dict[str, object]]:

"""

A\* search for the 8-puzzle.

Returns a dict with keys: path, moves, expanded\_nodes, or None if unsolvable.

"""

if start == goal:

return {"path": [start], "moves": 0, "expanded\_nodes": 0}

if not is\_solvable(start):

return None

# Each heap item: (f, g, tie, state)

heap: List[Tuple[int, int, int, State]] = []

tie = count()

g0 = 0

h0 = heuristic(start, goal)

heapq.heappush(heap, (g0 + h0, g0, next(tie), start))

best\_g: Dict[State, int] = {start: 0}

parents: Dict[State, State] = {}

expanded = 0

while heap:

f, g, \_, state = heapq.heappop(heap)

expanded += 1

if state == goal:

path = reconstruct\_path(parents, state)

return {

"path": path,

"moves": len(path) - 1,

"expanded\_nodes": expanded,

"move\_sequence": path\_to\_moves(path),

}

for nb, \_mv in neighbors(state):

new\_g = g + 1

if nb not in best\_g or new\_g < best\_g[nb]:

best\_g[nb] = new\_g

parents[nb] = state

h = heuristic(nb, goal)

heapq.heappush(heap, (new\_g + h, new\_g, next(tie), nb))

return None # Shouldn't happen for solvable instances

def print\_board(state: State) -> None:

"""Pretty-print a 3x3 board."""

for r in range(3):

row = state[3 \* r : 3 \* r + 3]

print(" ".join(str(x) if x != 0 else "·" for x in row))

print()

def parse\_state(s: str) -> State:

"""

Parse a state from a string like "1,2,3,4,0,6,7,5,8" or "123406758".

"""

s = s.strip().replace(" ", "")

if "," in s:

nums = [int(x) for x in s.split(",")]

else:

if len(s) != 9 or not s.isdigit():

raise ValueError("Provide 9 digits including 0 for the blank, or 9 comma-separated numbers.")

nums = [int(ch) for ch in s]

if len(nums) != 9 or set(nums) != set(range(9)):

raise ValueError("State must contain exactly the numbers 0..8 once each.")

return tuple(nums) # type\_ignore[return-value]

def main(start\_state\_str: str = "1,2,3,4,0,6,7,5,8", goal\_state\_str: str = "1,2,3,4,5,6,7,8,0", heuristic\_name: str = "manhattan", show\_path: bool = True):

start = parse\_state(start\_state\_str)

goal = parse\_state(goal\_state\_str)

# Reset goal positions cache if using a different goal

\_goal\_positions.cache\_clear()

heur: Callable[[State, State], int] = manhattan\_distance if heuristic\_name == "manhattan" \

else misplaced\_tiles

result = a\_star(start, heuristic=heur, goal=goal)

if result is None:

print("This puzzle is unsolvable.")

return

path: List[State] = result["path"] # type\_ignore[assignment]

moves: int = result["moves"] # type\_ignore[assignment]

expanded: int = result["expanded\_nodes"] # type\_ignore[assignment]

seq: str = result.get("move\_sequence", "") # type\_ignore[assignment]

print(f"Solved in {moves} moves using {heuristic\_name} (expanded {expanded} states).")

print(f"Move sequence: {seq}")

if show\_path:

print("\nPath:")

for i, st in enumerate(path):

print(f"Step {i}:")

print\_board(st)

main()

