Daniel casanova
Aula 4

Estado inicial: O estado inicial do n-puzzle é uma configuração gerada aleatoriamente das peças. Pode ser representado como uma lista de listas, onde cada lista interna representa uma linha do quebra-cabeça e a peça em branco é representada pelo número 0.

Ações: As ações possíveis em qualquer estado determinado são determinadas pela posição do ladrilho em branco. O ladrilho em branco pode ser movido para cima, para baixo, para a esquerda ou para a direita se houver um ladrilho adjacente nessa direção. Essas ações podem ser representadas como um dicionário onde as chaves são os nomes das ações e os valores são listas que representam a mudança na posição do ladrilho em branco quando a ação é aplicada.

Modelo de transição: O modelo de transição define como o estado muda quando uma ação é aplicada. Quando uma ação é aplicada a um estado, o ladrilho em branco é movido na direção especificada pela ação e o ladrilho que estava anteriormente nessa posição é movido para a posição anteriormente ocupada pelo ladrilho em branco.

Teste de meta: O teste de meta verifica se um determinado estado é o estado de meta. O estado de objetivo para o quebra-cabeça n é uma configuração em que todas as peças são organizadas em ordem crescente da esquerda para a direita e de cima para baixo, com a peça em branco no canto inferior direito.

Custo do caminho: O custo do caminho para o quebra-cabeça n pode ser definido como um custo constante para cada ação. Por exemplo, cada ação pode ter um custo de 1.

```
import random
from collections import deque

# define the size of the puzzle (n)
n = 3

# define the goal state of the puzzle
goal_state = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

# define the actions that can be taken as a dictionary
actions = {
    'up': [-1, 0],
    'down': [1, 0],
    'left': [0, -1],
    'right': [0, 1]
}

def generate_initial_state(n):
    """Generates a random initial state for the n-puzzle"""
    state = [i for i in range(n*n)]
```

```
    random.shuffle(state)
    return [state[i:i+n] for i in range(0, len(state), n)]

def list_actions(state):
    """Lists the possible actions that can be taken in a given state"""
    # find the position of the blank tile (0)
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                x, y = i, j
                break

    # check which actions are possible
    possible_actions = []
    for action in actions:
        new_x, new_y = x + actions[action][0], y + actions[action][1]
        if new_x >= 0 and new_x < len(state) and new_y >= 0 and new_y < len(state[0]):
            possible_actions.append(action)

    return possible_actions

def apply_action(state, action):
    """Applies an action to a given state and returns the resulting state"""
    # find the position of the blank tile (0)
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                x, y = i, j
                break

    # apply the action
    new_x, new_y = x + actions[action][0], y + actions[action][1]
    new_state = [row[:] for row in state]
    new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]

    return new_state

def is_goal(state):
    """Checks if a given state is the goal state"""
    return state == goal_state

def cost():
    """Returns the constant cost of an action"""
    return 1

def solve(initial_state):
    """Solves the n-puzzle using breadth-first search"""
    # create a queue to store the states to be explored
```

```
    queue = deque([(initial_state, [])])

    # create a set to store the explored states
    explored = set()

    # loop until the queue is empty
    while queue:
        # get the next state from the queue
        state, path = queue.popleft()

        # check if the state is the goal state
        if is_goal(state):
            return path

        # add the state to the set of explored states
        explored.add(tuple(map(tuple, state)))

        # get the possible actions for the current state
        possible_actions = list_actions(state)

        # loop through the possible actions
        for action in possible_actions:
            # apply the action to get a new state
            new_state = apply_action(state, action)

            # check if the new state has already been explored
            if tuple(map(tuple, new_state)) not in explored:
                # add the new state to the queue
                queue.append((new_state, path + [action]))

# generate a random initial state
initial_state = generate_initial_state(n)

# print the initial state
print("Initial state:")
for row in initial_state:
    print(row)

# solve the puzzle and print the solution
solution = solve(initial_state)
print("Solution:")
for action in solution:
    print(action)
```

peguei uma saida de exemplo para demonstrar

Initial state:
[6, 2, 8]
[1, 4, 7]
[3, 5, 0]
Solution:
up
up
left
down
down
right
up
left
up
left
down
down
right
up
up
left
down
right
down
left
up
up