Richard Šléher, ID: 120890, ak.r.:2023/2024

Slovenská technická univerzita v Bratislave

Fakulta informatiky a informačných technológií

Umelá inteligencia

Zadanie 1: Riešenie 8-hlavolamu použitím A* algoritmu a porovnanie dvoch heuristík

Obsah

Zadanie

Úlohou je implementovanie A* algoritmu na vyriešenie 8-hlavolamu, kde sa nachádza 8 číslic a jedno prázdne políčko, ktoré slúži na pohyb po ploche. Ďalej treba porovnať dve heuristiky pričom sa prvá zakladá na políčkach, ktoré nie sú v koncovej pozícii a druha je založená na súčte vzdialeností jednotlivých políčok od ich cieľovej pozície.

Opis riešenia

Vytvoríme si 2 zoznamy, v jednom budú uzly, ktoré už prejdené boli a v druhom budú zoradené neprejdené uzly podľa ich f hodnoty. Táto hodnota sa počíta sčítaním hodnoty heuristiky a hĺbkou uzla. Na začiatku pridáme do otvoreného zoznamu počiatočný stav. Ak je heuristika 0 tak sme v cieľovej pozícii a algoritmus sa skončí. Zistíme možné pohyby a vypočítame ich f hodnotu. Overíme či nový susedný stav je nie je v zozname s už prejdenými hodnotami. Ak nie je tak uzol vytvoríme a pridáme do zoznamu s ešte neprejdenými uzlami. Ak je uzol v nespracovaných tak skontrolujeme, či sme nenašli lepší. Skontrolujeme, či otvorený zoznam nie je prázdny ak je tak sme nenašli riešenie inak pokračujeme.

A* algoritmus

```
# Function to implement A* algorithm to solve the 8-puzzle problem
def a_star(initial_state, final_state, heuristic):
    open_list = []  # Contains the nodes that are to be visited
    closed_list = set()  # Contains unique states only

    if heuristic == "total":
        node = Node(initial_state, None,0, 0,
total_length(initial_state, final_state))
    else:
```

```
        node = Node(initial_state, None,0, 0,
misplaced_heuristic(initial_state, final_state))

    node.f = node.h + node.g
    # Add the initial state to the open list
    heapq.heappush(open_list, node)

    while open_list:

        # Removes the first element from the open list and removes it
        current_node = heapq.heappop(open_list)

        # Check if the current node is the goal state
        if current_node.h == 0:
            path = get_path(current_node)
            return path

        # Get possible moves
        possible_neighbors = get_neighbors(current_node.state)

        # Add the current state to the closed list
        closed_list.add(tuple(map(tuple, current_node.state)))

        # Create nodes for each possible move
        for neighbor in possible_neighbors:

            # Check if the state is in the closed list
            if tuple(map(tuple, neighbor)) not in closed_list:

                if heuristic == "total":
                    child_node = Node(neighbor, current_node,
current_node.g + 1, 0, total_length(neighbor, final_state))
                else:
                    child_node = Node(neighbor, current_node,
current_node.g + 1, 0, misplaced_heuristic(neighbor, final_state))

                child_node.f = child_node.g + child_node.h

                heapq.heappush(open_list, child_node)

            else:
                for node in open_list:
                    # If the state already exists
                    if node.state == neighbor:
                        if node.f > current_node.f:
                            node.g = current_node.g + 1
```

```
                            node.f = node.g + node.h
                            node.parent = current_node

    return None
```

Obsah uzla

V uzloch som uchovával stav, hodnotu f, rodiča, hĺbku a hodnotu heuristiky.

```
# Define a class for nodes in the search tree
class Node:
    def __init__(self, state, parent,g, f, h):
        self.state = state
        self.parent = parent
        self.g = g
        self.f = f
        self.h = h
```

Operátory

Operátory sú zakomponované vo funkcii, ktorá zisťuje susedné stavy.

```
# Function to get neighbors
def get_neighbors(board):
    neighbors = []
    empty_row, empty_col = find_empty_space(board)
    # RIGHT LEFT DOWN UP
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for d_row, d_col in moves:
        # New position of the empty space
        new_row, new_col = empty_row + d_row, empty_col + d_col
        # Check if the new position is in the board
        if 0 <= new_row < len(board) and 0 <= new_col < len(board):
            new_board = copy.deepcopy(board)
            # Swap the empty space with the tile
            new_board[empty_row][empty_col],
new_board[new_row][new_col] = new_board[new_row][new_col],
new_board[empty_row][empty_col]
            neighbors.append(new_board)

    return neighbors
```

Heuristiky

Heuristika 1, kde sa zisťuje počet čísel, ktoré nie sú na svojom mieste.

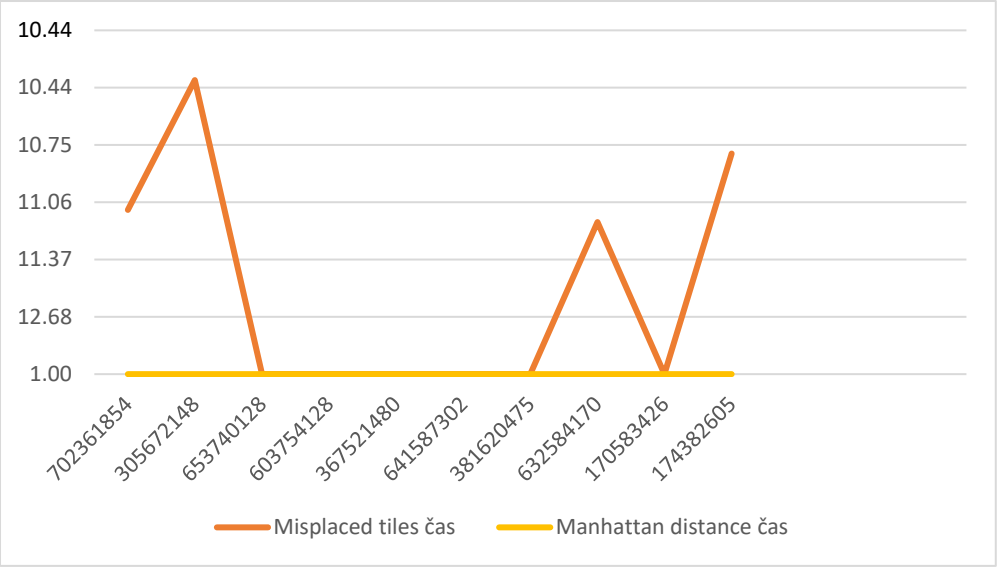
```
# Function to calculate the heuristic for misplaced tiles
def misplaced_heuristic(board, target):
    count = 0
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] != target[i][j]:
                count += 1
    return count
```

Heuristika 2, kde sa vypočíta celková vzdialenosť rozdielov políčkom na nesprávnom mieste.

```
# Function to calculate the heuristic for total distance of misplaced
tiles
def total_length(current, target):

    total_distance = 0

    for i in range(len(current)):
        for j in range(len(current)):
            wanted = current[i][j]
            if wanted != 0:
                # Find the position of the current tile in the target
board
                for k in range(len(target)):
                    for l in range(len(target)):
                        if wanted == target[k][l]:
                            # Calculate the distance between the
current tile and the target tile
                            total_distance += abs(i - k) + abs(j - l)

    return total_distance
```

Testovanie

Vygeneroval som dve polia 10 matíc a porovnával som rýchlosti ich tvorenia dvomi heuristikami .



Graf porovnáva rýchlosti riešenia dvomi heuristikami

Porovnanie výsledkov

Heuristika, ktorá počíta celkové vzdialenosti je rýchlejšia ako tá, ktorá počíta nesprávne polohy políčok. Testovanie sa vykonávalo na maticiach 3x3, ktoré boli náhodne generované a riešiteľné.