**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** 3](#_Toc147767659)

[**Opis riešenia** 3](#_Toc147767660)

[**A\* algoritmus** 3](#_Toc147767661)

[**Obsah uzla** 5](#_Toc147767662)

[**Operátory** 5](#_Toc147767663)

[**Heuristiky** 5](#_Toc147767664)

[**Testovanie** 6](#_Toc147767665)

[**Porovnanie výsledkov** 7](#_Toc147767666)

# **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.