**Projekt 1**

**Prehľadávanie stavového priestoru**

**Laura Fulajtárová**

Fakulta informatiky a informačných technológií STU

xfulajtarova@stuba.sk

ID: 120782

UI – Utorok 14:00

5.10.2023

Contents

[Riešený problém 3](#_Toc147508074)

[Opis riešenia 3](#_Toc147508075)

[A\* algoritmus 3](#_Toc147508076)

[Heuristika 1 - Wrong tiles 4](#_Toc147508077)

[Heuristika 2 - Manhattan distance 4](#_Toc147508078)

[Rotácia puzzle vzhľadom na polohu prázdneho políčka 5](#_Toc147508079)

[Generovanie možných ťahov a pridávanie uzlov do haldy 5](#_Toc147508080)

[Reprezentáciu údajov problému 6](#_Toc147508081)

[Spôsob testovania a výsledky experimentov 6](#_Toc147508082)

[Ľahká obtiažnosť 6](#_Toc147508083)

[Stredná obtiažnosť 7](#_Toc147508084)

[Ťažká obtiažnosť 8](#_Toc147508085)

[Neriešiteľné príklady 9](#_Toc147508086)

[Zhodnotenie riešenia a dosiahnutých výsledkov 10](#_Toc147508087)

[Podľa času 10](#_Toc147508088)

[Podľa krokov 10](#_Toc147508089)

[Podľa vytvorených uzlov 11](#_Toc147508090)

[Používateľská príručka na spustenie programu 11](#_Toc147508091)

[Použitá literatúra 12](#_Toc147508092)

# Riešený problém 2 úloha e)

Zadanie spočíva v riešení 8-hlavolamu na hracej doske o veľkosti MxM, pričom som testovala problémy s doskami o rozmeroch 3x3 a 4x4. Hlavolam pozostáva z osmi číselných políčok a jedného prázdneho miesta. Hráč môže presúvať políčka nahor, nadol, vľavo alebo vpravo, avšak iba v smere prázdneho miesta. Začiatočný a cieľový stav hlavolamu sú vopred definované, a úlohou je nájsť optimálnu postupnosť krokov, ktorá vedie z počiatočného stavu k cieľovému.

Na riešenie tohto problému je použitý A\* algoritmus spolu s heuristickými funkciami. Tieto heuristiky poskytujú doplnkové informácie o vzdialenosti medzi aktuálnym stavom a cieľovým stavom. Konkrétne heuristiky zahŕňajú počet políčok, ktoré sa nenachádzajú na svojich správnych pozíciách, a súčet vzdialeností jednotlivých políčok od ich cieľových pozícií. Tieto heuristiky pomáhajú rozhodnúť, akým smerom sa vydať, aby sa čo najefektívnejšie dosiahol cieľový stav.

Konkrétny príklad:

Obrázok, na ktorom je text, snímka obrazovky, písmo, číslo

Automaticky generovaný popis Obrázok, na ktorom je text, snímka obrazovky, písmo, číslo

Automaticky generovaný popis

V rámci tohto problému existuje konkrétna postupnosť krokov, ktorá vedie k riešeniu, napríklad kroky DOLE, DOPRAVA a znovu DOLE. Navrhovaná metóda algoritmu musí byť schopná identifikovať a zabezpečiť túto konkrétnu postupnosť krokov prostredníctvom výpočtu heuristík a porovnávania krokov.

# Opis riešenia

## A\* algoritmus

A\* je algoritmus na hľadanie optimálnej cesty v grafe. Využíva otvorený zoznam na uchovávanie nepreskúmaných uzlov a uzavretý zoznam na preskúmané uzly. Priorita uzlov sa určuje kombináciou ceny dosiahnutej cesty (g) a heuristického odhadu (h), f = g + h.

**Moja implementácia:**

Najskôr som si inicializovala a prípadne vynulovala všetky potrebné premenné. Na uchovávanie uzlov a ich funkcií som využívala otvorený a uzavretý zoznam (open a closed list).

Na začiatku som pridala počiatočnú konfiguráciu hracej plochy do prioritnej fronty (priority queue) pomocou štruktúry dát s názvom "heapq". Táto štruktúra "heapq" umožňuje efektívne spravovať prioritné rady (min-heap), kde prvok s najnižšou prioritou je na vrchole. Funkcia heappush mi umožnila pridávať uzly do tohto min-heapu podľa priority.

Potom som pomocou cyklu while postupne pridávala možné riešenia do haldy (heap) a monitorovala som počet vytvorených uzlov pre prípad, že by som chcela program ukončiť v prípade vysokého času.

Keď som pridávala deti do haldy, použila som funkciu heappush, ktorá umožňuje vložiť prvok do haldy tak, aby sa zachovala jej vlastnosť min-heapu.

Ak som objavila riešenie vo funkcii insert(node, board\_end, open\_list, n, choice), zaznamenala som počet vytvorených uzlov, vypísala som cestu k riešeniu pomocou funkcie print\_solution\_path(solution\_node, choice) a ukončila som hlavnú funkciu.

def astar(board\_start, board\_end, n, choice):

    global closed\_set, solution, num\_of\_moves\_h1, num\_of\_moves\_h2, num\_of\_nodes\_h1, num\_of\_nodes\_h2

    closed\_set = set()

    solution = None

    start\_node = Node(board\_start)

    open\_list = []

    node\_count = 0

    heapq.heappush(open\_list, (start\_node.f, start\_node))

    while open\_list:

        \_, current\_node = heapq.heappop(open\_list)

        node\_count += 1

        if node\_count >= 50000:

            print("Reached 50000 nodes, exiting...")

            if choice == 1:

                num\_of\_moves\_h1 = 0

            elif choice == 2:

                num\_of\_moves\_h2 = 0

            break

        insert(current\_node, board\_end, open\_list, n, choice)

        if solution:

            if choice == 1:

                num\_of\_nodes\_h1 = node\_count

            elif choice == 2:

                num\_of\_nodes\_h2 = node\_count

            print\_solution\_path(solution, choice)

            break

## Heuristika 1 - Wrong tiles

Jednou z heuristík, ktorú som použila, je "počet zle umiestnených políčok." Táto heuristika spočíta, koľko políčok nie je na správnom mieste vzhľadom na cieľový stav.

Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn

Automaticky generovaný popis Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn

Automaticky generovaný popis h1=5



def heuristic\_1\_wrong\_tiles(board, board\_end):

    count = 0

    for i in range(len(board)):

        if board[i] != board\_end[i] and board[i] != 0:

            count += 1

    return count

## Heuristika 2 - Manhattan distance

Druhou použitou heuristikou je "Manhattanova vzdialenosť." Táto metrika meria najkratšiu vzdialenosť medzi dvoma bodmi na mriežke podľa počtu krokov, ktoré sú potrebné vo vodorovnom a zvislom smere, aby sa dostali z jedného bodu do druhého.

# Function to calculate the Manhattan distance for a single tile

def manhattan\_distance(tile, current\_position, goal\_position, n):

    if tile == 0:

        return 0

    tile -= 1

    current\_row, current\_col = current\_position // n, current\_position % n

    goal\_row, goal\_col = goal\_position // n, goal\_position % n

    return abs(current\_row - goal\_row) + abs(current\_col - goal\_col)

# Function to calculate the sum of Manhattan distances for all tiles

def heuristic\_2\_tiles\_distance(board, goal, n):

    distance = 0

    for i in range(len(board)):

        tile\_value = board[i]

        goal\_position = goal.index(tile\_value)

        distance += manhattan\_distance(tile\_value, i, goal\_position, n)

    return distance

Obrázok, na ktorom je text, snímka obrazovky, písmo, číslo

Automaticky generovaný popis Obrázok, na ktorom je text, snímka obrazovky, písmo, číslo

Automaticky generovaný popis

distance for single tile 5 = 3

distance for every tile: 35 + 34 + 31 + 12 + 33 + 27 + 16 + 48 = 20

## Rotácia puzzle vzhľadom na polohu prázdneho políčka

Pre rotáciu hracej dosky som vytvorila štyri funkcie: "hore", "dole", "vpravo" a "vľavo". Každá z týchto funkcií má za úlohu vymeniť prázdne políčko, ktoré je označené 0, so svojím susedom. Tieto funkcie sú totožné okrem časti kódu, ktorá je vyznačená na obrázku. V tejto časti kontrolujeme hraničné podmienky, kde sa prázdne políčko nemôže posunúť, a tiež určujeme, s ktorým susedom sa má vymeniť, vzhľadom na smer, v ktorom chceme otáčať puzzle. Každá z týchto funkcií vytvára kópiu hracej plochy, aby sa zachoval pôvodný stav, a potom vykoná presun prázdneho políčka. Po vykonaní pohybu funkcia vráti nový stav hracej plochy.

def up(board, n):

    board\_copy = copy.deepcopy(board)

    for i in range(len(board\_copy)):

        if board\_copy[i] == 0:

            if i - n >= 0:

                board\_copy[i], board\_copy[i - n] = board\_copy[i - n], board\_copy[i]

                return board\_copy

    return board\_copy

down

left

right

            if i + n < len(board\_copy):

                board\_copy[i], board\_copy[i + n] = board\_copy[i + n], board\_copy[i]

            if i % n != 0:

                board\_copy[i], board\_copy[i - 1] = board\_copy[i - 1], board\_copy[i]

            if (i + 1) % n != 0:

                board\_copy[i], board\_copy[i + 1] = board\_copy[i + 1], board\_copy[i]

## Generovanie možných ťahov a pridávanie uzlov do haldy

Na vloženie detí do prioritnej fronty som vytvorila funkciu s názvom "insert". Najskôr kontrolujeme, či aktuálna hracia doska nie je naším konečným cieľom. Ak áno, uložíme ju ako riešenie a ukončíme vyhľadávanie. Potom kontrolujeme, či je aktuálna hracia doska už v uzavretých stavoch. Ak áno, preskočíme ju a nepokračujeme v jej ďalšom spracovaní. V opačnom prípade ju pridáme do množiny uzavretých stavov.

Ďalej funkcia generuje všetky možné rotácie hracej dosky, teda otáčanie doľava, doprava, nahor a nadol. Tieto vytvorené dosky skúmame tak, že ich porovnávame s aktuálnou doskou, pretože v prípade, že by posun bol mimo hracej dosky, stav by zostal nezmenený, a preto môžeme pokračovať len s platnými pohybmi.

Pre každé takto vytvorené dieťa vytvoríme uzol s príslušným rozložením dosky, operátorom, aktualizovanou hĺbkou, vypočítanou funkciou pre danú heuristiku a odkazom na jeho predchodcu, tzv. jeho rodiča.

Ďalej kontrolujeme, či takto vytvorené dieťa už nie je v open liste. Ak nie je, pridáme ho do priority queue vzhľadom na jeho f hodnotu pomocou funkcie "heappush".

def insert(node, board\_end, open\_list, n, choice):

    if is\_same(node.board, board\_end):

        global solution

        solution = copy.deepcopy(node)

        return None

    if tuple(node.board) in closed\_set:

        return node

    closed\_set.add(tuple(node.board))

    l\_ch = left(node.board, n)

    r\_ch = right(node.board, n)

    u\_ch = up(node.board, n)

    d\_ch = down(node.board, n)

    if not is\_same(l\_ch, node.board):

        child = Node(

            l\_ch,

            "left",

            node.depth + 1,

            parent=node,

            f=node.depth + 1 + calculate\_heuristic(choice, l\_ch, board\_end, n),

        )

        if child.board not in open\_list:

            heapq.heappush(open\_list, (child.f, child))

    ....... toto opakujeme pre r\_ch, u\_ch, d\_ch len s iným operandom

    return node

# Reprezentáciu údajov problému

Pre reprezentáciu stavu a nájdenú cestu som vytvorila triedu Node. V tejto triede som uchovávala rôzne typy údajov, ako je rozloženie hracej dosky, operátor, ktorým sme sa dostali k tomuto stavu (ľavý, pravý, hore, dole), hĺbku uzla (na výpočet funkcie "f"), samotnú funkciu "f" a odkaz na rodiča (na neskoršie vypísanie cesty).

class Node:

    def \_\_init\_\_( self, board=[], operand=None, depth=0, parent=None, f=None,):

        self.board = board

        self.operand = operand

        self.depth = depth

        self.f = f

        self.parent = parent

# Spôsob testovania a výsledky experimentov

Určila som štyri rôzne scenáre na testovanie, pričom som brala do úvahy obtiažnosť a riešiteľnosť problému.

## Ľahká obtiažnosť

Na začiatok som oba prístupy testovala na veľmi jednoduchých príkladoch s maximálne 5 krokmi, pri ktorých obe heuristiky rýchlo našli riešenie. V tomto prípade sa heuristika 2, založená na Manhattanských vzdialenostiach, v časovej náročnosti ukázala ako výhodnejšia, avšak rozdiel nebol tak výrazný ako pri iných skúmaných scenároch. Obidve heuristiky viedli k rovnakému počtu krokov a taktiež zväčša aj k rovnakému počtu vytvorených uzlov.

|  |  |
| --- | --- |
| **3x3 BOARD** | |
| Obrázok, na ktorom je text, snímka obrazovky, písmo, číslo  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo, číslo  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | |
| **4x4 BOARD** | |
|  |  |
|  |  |
|  | |

## Stredná obtiažnosť

V prípade stredne náročných príkladov s približne dvadsiatimi krokmi excelovala heuristika 2 v rýchlosti a efektívnosti vzhľadom na počet vytvorených uzlov, pričom cesta riešenia zostávala totožná.

|  |  |
| --- | --- |
| **3x3 BOARD** | |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, jedálny lístok, dizajn  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, dizajn  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | |
| **4x4 BOARD** | |
|  |  |
|  |  |
|  | |

## Ťažká obtiažnosť

Náročné príklady s približne tridsiatimi krokmi predstavovali výzvu, pri ktorej prvá heuristika, ktorá počet vytvorených uzlov prekročila, musela byť prerušená. Naopak, heuristika 2 zvládla tento problém v akceptovateľnom čase s primeraným početom vytvorených uzlov.

|  |  |
| --- | --- |
| **3x3 BOARD** | |
| Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn  Automaticky generovaný popis |
| Obrázok, na ktorom je text, písmo, snímka obrazovky  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, dizajn  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | |
| **4x4 BOARD** | |
|  |  |
|  |  |
|  | |

## Neriešiteľné príklady

Posledným skúmaným scenárom boli príklady, ktoré obidve heuristiky nedokázali vyriešiť, označované ako neriešiteľné. V tomto prípade sme porovnali rýchlosť, s akou oba prístupy zistili neriešiteľnosť problému, pričom heuristika 2 opäť ukázala svoju spoľahlivosť.

|  |  |
| --- | --- |
| **3x3 BOARD** | |
| Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo, dizajn  Automaticky generovaný popis |
| Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis | Obrázok, na ktorom je text, snímka obrazovky, písmo  Automaticky generovaný popis |
| Obrázok, na ktorom je text, písmo, snímka obrazovky  Automaticky generovaný popis | |
| **4x4 BOARD** | |
|  |  |
|  |  |
|  | |

# Zhodnotenie riešenia a dosiahnutých výsledkov

V kontexte mojho riešenia tejto úlohy môžem konštatovať, že A\* algoritmus sa ukázal ako výnimočne účinný a rýchly v porovnaní s prístupom nahodilého výberu krokov. Jeho schopnosť využívať heuristiky pre rýchle rozhodovanie o výbere uzlov v procese hľadania riešenia je pozoruhodná. Avšak rozhodnutie o vhodnej heuristike zásadným spôsobom ovplyvnilo dosiahnuté výsledky.

V tomto kontexte vynikla heuristika 2, založená na Manhattanských vzdialenostiach, ako veľmi rýchla a efektívna voľba pri hľadaní riešenia hracej dosky. Táto heuristika často generovala menší počet uzlov v stromovom vyhľadávacom priestore a zásadne prispela k rýchlejšiemu približovaniu sa k riešeniu. Zaujímavé je, že postupnosti krokov na dosiahnutie riešenia ostali totožné, ale heuristika 2 sa ukázala výrazne efektívnejšou.

Obmedzenie na maximálny počet vytvorených uzlov (50000) viedlo k tomu, že heuristika 1 (počet nesprávne umiestnených políčok) v niektorých prípadoch neobjavila správnu cestu k riešeniu. Naopak, heuristika 2 bola schopná nájsť riešenie, hoci to trvalo trochu dlhšie, čím potvrdila svoju spoľahlivosť pri hľadaní optimálneho riešenia.

Na základe týchto pozorovaní by som bez váhania odporučila použitie heuristiky 2 (Manhattanská vzdialenosť) v tejto úlohe, keďže sa ukázala ako rýchlejšia, efektívnejšia a spoľahlivejšia vo voľbe heuristiky pre A\* algoritmus pri riešení hracej dosky.

Nasledujúce grafy boli zostavené podľa získaných informácií v časti [Spôsob testovania a výsledky experimentov](#_Spôsob_testovania_a).

## Podľa času

|  |  |
| --- | --- |
|  |  |
|  |

## Podľa krokov

|  |  |  |
| --- | --- | --- |
|  |  |  |

## Podľa vytvorených uzlov

|  |  |  |
| --- | --- | --- |
|  |  |  |

# Používateľská príručka na spustenie programu

Moja práca sa skladá z dvoch súborov: main.py a game.py. V súbore game.py som naprogramovala logiku projektu a obsahuje rôzne metódy a triedy pre vytvorenie uzlov, algoritmus A\*, vloženie detí, výpis cesty a ďalšie. Všetky podstatné informácie som taktiež uviedla v dokumentácii v časti s názvom [Opis riešenia](#_Opis_riešenia).

Súbor main.py slúži na spustenie celého programu. Navrhla som jednoduché rozhranie pre interakciu s programom. Po spustení programu vás program privíta a ponúkne vám tri základné možnosti: ukončiť hru, pokračovať s vami navrhnutým rozložením počiatočnej a cieľovej hracej dosky alebo automaticky vygeneruje náhodné rozloženia. Taktiež od vás program očakáva, aby ste zadali počet stĺpcov a riadkov hracej dosky, pričom odporúčam veľkosť 3x3, ale program bol testovaný aj na hracích doskách o veľkosti 4x4. Následne program zobrazí hracie dosky, s ktorými bude pracovať, a začne ich riešiť podľa heuristiky 1 alebo heuristiky 2. Po nájdení riešenia program zobrazí cestu a počet krokov. Taktiež poskytne porovnanie týchto heuristík vzhľadom na čas, počet krokov k správnemu riešeniu a počet uzlov, ktoré boli potrebné na vytvorenie riešenia. Po týchto štatistikách si môžete hru zahrať znova alebo ju ukončiť stlačením klávesy "q".

# Použitá literatúra

1. <https://www.simplilearn.com/tutorials/artificial-intelligence-tutorial/a-star-algorithm>
2. <https://www.geeksforgeeks.org/heap-queue-or-heapq-in-python/>