Vysoké učení technické v Brně
Ústav automatizace a informatiky

Technická zpráva

Implementace algoritmu pro úlohu 3x3 puzzle

Autor: Lukáš Podmela

Datum: 22.12.2024

1 Úvod

Tato technická zpráva se zabývá implementací algoritmu pro řešení úlohy 3x3 puzzle (Lišák). Hra 3x3 puzzle je klasický problém, kdy na čtvercové mřížce 3x3 posouváme dlaždici za účelem předem stanoveného uspořádání. Cílem tohoto projektu je vytvoření algoritmu pro nalezení ideálně co nejmenšího množství pohybů pro dosažení cílového stavu ze stavu počátečního. Tato úloha je názorný algoritmizační problém, pro demonstraci základních vyhledávajících algoritmů, jako je BFS, DFS nebo například pro využití heuristiky v A* algoritmu.

2 Teoretický popis řešené úlohy

Jak již bylo zmíněno, 3x3 puzzle spočívá v přesouvání dlaždic do cílového stavu. Ve čtvercové mřížce se nachází celkem 8 dlaždic a jedno prázdné místo sloužící pro pohyb. V této úloze je pohyb umožněn pouze takovým dlaždicím, jejichž strana sousedí s prázdným místem. Pohyb probíhá výměnou pozice dlaždice s prázdným místem. Cílem je dosažení minimálního množství tahů.[3]

V implementaci této úlohy jsou jednotlivé dlaždice reprezentovány číslicemi 1-8, přičemž prázdné místo 0. Cílový stav je definován posloupností po řádcích 123, 456, 780, jak je zobrazeno na Obrázku 1.
V rámci této úlohy existuje 181440 možných stavů, ale pouze polovina je řešitelná. Řešitelnost této úlohy je závislá na počtu inverzí (počet párů dlaždic, kde větší číslo je před menším), který musí být sudý.[1]

Nejnáročnější počáteční stav této úlohy, který má zároveň řešení, je posloupnost čísel po řadách: 867, 254, 301, kdy nejmenší počet tahů pro vyřešení této úlohy je 31.[2]

1	2	3
4	5	6
7	8	0

Obrázek 1: Cílový stav

3 Metody pro řešení úlohy

Pro řešení úlohy 3x3 puzzle existuje několik algoritmů, které je možné využít. Nejčastěji se využívá metod, které nám umožní prozkoumat všechny možné pohyby v prostoru stavů, dokud nenajdou cestu k cílovému stavu.

- **BFS** – pomocí prohledávání do šířky (BFS algoritmus) je zaručeno nalezení optimálního řešení s vyšší časovou náročností. Prohledává všechny možné stavy rovnoměrně po jednotlivých úrovních.
- **DFS** – algoritmus prohledávání do hloubky prohledává nejdále do hloubky před tím, než se vrátí k prozkoumání dalších možností. Algoritmus není optimální a nemusí najít nejkratší cestu, efektivita lze najít v prozkoumávání jednotlivých cest.
- **A*** - A star je heuristický algoritmus, který je použit v této implementaci 3x3 puzzle. Algoritmus využívá principů Dijkstrova algoritmu, rozšířený o heuristický prvek.
- **Manhattanova vzdálenost** - heuristika užita v této implementaci. Metoda spočívá v součtu vzdáleností dlaždice k její cílové pozici na mřížce. Zvolen je pohyb s nejnižším součtem manhattanovy metriky, pohyb nesmí být shodný s předchozím stavem úlohy.

4 Popis vytvořeného SW

Vytvořený software implementuje algoritmus A star s Manhattanskou heuristikou v Pythonu. Pro tuto implementaci je v práci využito knihoven heapq pro práci s frontami a zároveň s knihovnami Tkinter pro vytvoření grafického rozhraní.

V přiloženém zdrojovém kódu jsou zpočátku iniciovány počáteční a koncová matice, zároveň jsou zde definovány možné tahy jednotlivých dlaždic v rámci matice. Zároveň jsou v počátcích kódu definovány jednoduché funkce využívané v dalších částech kódu, jako je funkce find-index pro nalezení indexu pro požadované číslo (nebo například prázdného pole). Funkce swap pro výměnu pozic dvou čísel. Funkce get-neighbours slouží pro získání všech možných stavů po uskutečnění všech možných pohybů, z těchto stavů následně probíhá výpočet manhattanské metriky.

Kód 1: Ověření řešitelnosti

```
1  def is_solvable(matrix):
2
3      # Flatten the matrix into a single list, ignoring the zero
4      flat_list = []
5      for row in matrix:
6          for num in row:
7              if num != 0:
8                  flat_list.append(num)
9
10     # Count the number of inversions
11     inversions = 0
12     for i in range(len(flat_list)):
13         for j in range(i + 1, len(flat_list)):
14             if flat_list[i] > flat_list[j]:
15                 inversions += 1
16
17     # Return True if the number of inversions is even
18     return inversions % 2 == 0
```

V Kódu 1 je zobrazena část kódu, ve které je implementována funkce na ověření řešitelnosti počítající počet inverzí. Parametrem této funkce je matice. V řádcích 4-8 je tato matice zploštěná do single listu, přičemž je ignorována prázdná dlaždice.

Z této zploštěné dlaždice je následně na řádcích 11-15 počítán counter inverzí kdy jsou porovnávány jednotlivé páry. Pokud je větší číslo před menším, proměnná inverze se zvětší o 1. Pokud inverze je sudá, funkce vrátí hodnotu True.

Kód 2: Implementace Manhattanské metriky

```
1 def manhattan(state):
2     man_num = 0
3     for number in range(1,9):
4         index_final = find_index(FINAL_MATRIX, number)
5         index_state = find_index(state, number)
6 
7         man_num += abs(index_final[0] - index_state[0]) + abs(
    index_final[1] - index_state[1])
8 
9     return man_num
```

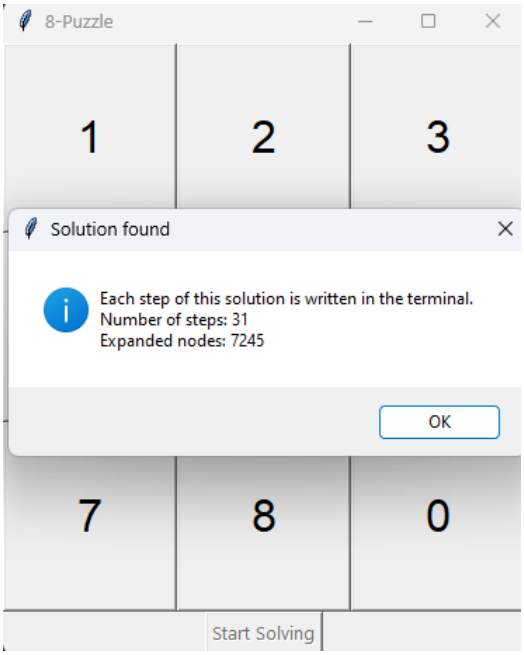
V kódu 2 je uvedená implementace manhattanské metriky, kdy parametrem je aktuální stav matice. Proměnné index-final a index-state reprezentují pozici jednotlivých čísel v rámci matice. Součet absolutních rozdílů pozic řádků a sloupců aktuální a finální matice přičte do counteru man-num hodnotu, jak lze vidět na řádku 7. Takto definovaná funkce vrací číselnou hodnotu proměnné man-num.

Kód 3: Implementace A-star algoritmu

```
1 def astar(start_matrix):
2 
3     if not is_solvable(start_matrix):
4         return None, 0
5 
6     expanded_nodes = 0
7     closed_set = {}
8     open_set = []
9     heapq.heappush(open_set,(0,start_matrix))
10    g_score = {str(start_matrix) : 0}
11    h_score = manhattan(start_matrix)
12    f_score = {str(start_matrix) : h_score}
13 
14    while open_set:
15        _, current = heapq.heappop(open_set)
16        expanded_nodes += 1
17 
18        if current == FINAL_MATRIX:
19            return path_reconstruct(closed_set, current), expanded_nodes
20 
21        for each in get_neighbours(current):
22            current_g_score = g_score[str(current)] + 1
23            neighbour_str = str(each)
24 
25            if neighbour_str not in g_score or current_g_score < g_score
    [neighbour_str]:
26                closed_set[neighbour_str] = current
27                g_score[neighbour_str] = current_g_score
28                f_score[neighbour_str] = current_g_score + manhattan(
    each)
29                heapq.heappush(open_set,(f_score[neighbour_str], each))
30    return None, expanded_nodes
```

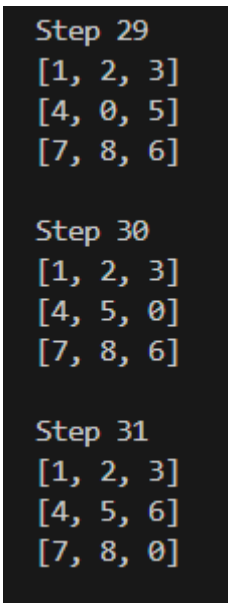
Kód 3 implementuje stěžejní A-star algoritmus. Parametrem tohoto algoritmu je úvodní stav matice. Pro ověření řešitelnosti je na samotný začátek zařazena funkce is-solvable. Na řádcích 6-12 inicializujeme proměnné důležité pro tento algoritmus. V rámci tohoto kódu je počítán počet prozkoumaných uzlů pro finální vyhodnocení, slovník již prozkoumaných stavů, open-set využívající prioritní frontu pro stavy potřebné k prozkoumaní, g-score zaznamenávající délku cesty od počátečního stavu, h-score využívající manhattanovy heuristiky a f-score, což značí celkovou hodnotu pro určení pořadí prozkoumání stavů $f = g + h$.
Počínaje řádkem 14 začíná hlavní cyklus algoritmu, který pokračuje dokud není fronta prázdná. V každém kroku odebíráme uzel se stejným stavem s nejnižším f-score. V případě, že je dosaženo cíle, znamená to nalezení cesty a funkce vrátí optimální cestu a počet prozkoumaných uzlů.
V opačném případě se pokračuje řádky 21-30, kde jsou prozkoumávány sousední stavy a porovnávány pomocí g-score. Pokud sousední stav ještě nebyl navštíven nebo existuje lepší cesta (nižší g-score) k sousednímu stavu, poté se aktualizuje closed-set, g-score a f-score. Pokud se algoritmus dostane nakonec a nenalezne cestu, vrátí None a počet prozkoumaných uzlů.

Další části bloků kódu, které jsou k nahlédnutí ve zdrojovém kódu pracují již pouze s poskytnutými daty z těchto funkcí. Jako například funkce path-reconstruct zachycující jednotlivé stavy než se matice dostala do cílového stavu. Nakonec je tato implementace algoritmu převedena do grafického rozhraní pomocí knihovny Tkinter. Grafické rozhraní je vytvořeno jako Class PuzzleVisualizer. Řešení je rovněž vyvedeno do terminalu pomocí print metody pro lepší názornější zobrazení jednotlivých kroků.



Obrázek 2: Finální grafické řešení

Na obrázku 2 lze vidět výslednou vizualizaci, kdy po dokončení animace řešení algoritmu vyskočí na uživatele pop-up dialog s výsledným počtem kroků a prozkoumaných uzlů. Jednotlivé kroky této animace postupují jednou za 1000ms.



Obrázek 3: Zobrazení v terminalu

Na závěr na Obrázku 3 je uvedena ukázka řešení v terminálu.

5 Závěr

Implementace A star algoritmu s manhattanskou heuristikou se jeví jako efektivní řešení pro úlohu A star algoritmu vzhledem ke kombinaci principů Dijkstrova algoritmu s vybranou heuristikou pro urychlení procesu hledání optimálního řešení. Cílem projektu bylo nalézt nejen správné řešení, ale také minimalizovat počet potřebných tahů, zvolené metody se ukázaly jako správné. V rámci implementace byla navržena vizualizace řešení pro lepší pochopení průběhu algoritmu a umožňuje sledovat proces v reálném čase.
Zvolené řešení lze optimalizovat pro řešení 4x4 Puzzle, kdy při testování tohoto problému je možné řešit jen problémy s počtem kroků do 51 s větší časovou náročností. Pro 3x3 puzzle by projekt bylo možné rozšířit o další algoritmizační metody.

Reference

[1] Edward Hordern. *Sliding Piece Puzzles*. Oxford University Press, 1986.

[2] w01fe. The hardest eight puzzle instances take 31 moves to solve, 2009. Accessed: 2024-12-24. URL: `https://w01fe.com/blog/2009/01/the-hardest-eight-puzzle-instances-take-31-moves-to-solve/`.

[3] Wikipedia contributors. Sliding puzzle, 2024. Accessed: 2024-12-24. URL: `https://en.wikipedia.org/wiki/Sliding_puzzle`.