Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Umelá inteligencia

Zadanie 1 – prehľadávanie stavového priestoru

Riešiteľ 8 hlavolamu – obojsmerné hľadanie

Petra Miková

Petra Miková
ID: 120852

Obsah

Petra Miková
ID: 120852

Obojsmerné vyhľadávanie

Ide o algoritmus prehľadávania grafu, pri ktorom sa súčasne uskutočňujú dve prehľadávania grafu – jedno smerom od počiatočného stavu, a druhé od koncového. Hľadanie sa ukončí ak sa tieto dve hľadania pretnú, teda narazia v grafe na spoločný vrchol (v našom prípade uzol stavu hlavolamu). Cieľom je nájdenie najkratšej vzdialenosti medzi vopred daným počiatočným a koncovým vrcholom. Tento prístup výrazne znižuje čas potrebný na prehľadanie grafu, a použitím heuristík ho vieme znížiť ešte viac.

Opis riešenia

Zadaním bolo použiť algoritmus obojsmerného hľadania pre nájdenie riešenia 8-hlavolamu. Môj program sa skladá z viacerých pomocných funkcií a jednej hlavnej funkcie, ktorá vykonáva spomínané obojsmerné vyhľadávanie. Celé zadanie je vypracované v jazyku Python.

Heuristika – Manhattanská vzdialenosť

V svojom riešení som v algoritme využila heuristiku Manhattanskej vzdialenosti. Táto heuristika odhaduje minimálny počet pohybov potrebných na dosiahnutie cieľového stavu z aktuálneho stavu. Vypočíta súčet vzdialeností medzi aktuálnou pozíciou každej časti hlavolamu a jej cieľovou pozíciou.

Konečné sortovanie stacku uzlov kombinuje cenu uzla (vzdialenosť aktuálneho uzla od počiatočného stavu) a práve túto heuristiku.

```
end_stack.sort(key=lambda node: node.cost + manhattan_distance(node.state, start_state))
```

Tento prístup pomáha algoritmu prioritizovať uzly ktoré sú bližšie k cieľovému stavu a hľadať tak najoptimálnejšie riešenie.

Opis dôležitých častí kódu

Uzol

Moja štruktúra uzlu pre reprezentovanie každého stavu vyzerá takto:

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def set_parent(self, parent):
        self.parent = parent

    def set_move(self, move):
        self.move = move

    def set_cost(self, cost):
        self.cost = costV
```

state: slúži na ukladanie aktuálneho stavu vo forme 2D listu (príklad: [[1,2,3],[4,5,6],[7,8,0])
parent: ukladá predchodcu tohto aktuálneho stavu
move: operácia, ktorou vykonaním sme sa dostali do aktuálneho stavu z pôvodného
cost: vzdialenosť od začiatku k tomuto uzlu, ktorú neskôr v programe využívam pri heuristike

Súčasťou tejto triedy Node sú už len základné set funkcie (settre).

Funkcie pre vykonanie pohybu

Pre vykonanie pohybu môj program obsahuje 3 krátke funkcie - find_blank_position, valid_move_check a apply_move.

find_blank_position: slúži na nájdenie prázdneho políčka v aktuálnom stave, teda v 2D liste hľadáme hodnotu 0, ktorá v mojom prípade toto políčko reprezentuje.

valid_move_check: jednoduchá funkcia ktorá overí, či je možné daný pohyb vykonať a zabráňuje teda vykonaniu nevalídnych krokov, napríklad posun dole ak je prázdne políčko na najsposnejšej vrstve apod. Volá sa priamo vo funkcii vykonávajúcej algoritmus hľadania.

apply_move: funkcia, ktorá vracia nový stav po vykonaní operácie pohybu. Zavolá si už spomínanu funkciu pre nájdenie prázdneho políčka, vytvorí kópiu stavu a aplikuje na nej daný posun.

Funkcia pre výpočet Manhattanskej vzdialenosti

```
def manhattan_distance(state, goal_state):
    distance = 0
    n = len(state)
    for i in range(n):
        for j in range(n):
            if state[i][j] != 0:
                current_position = (i, j)
                goal_value = state[i][j]
                goal_position = [(x, y) for x in range(n) for y in range(n)
if goal_state[x][y] == goal_value][0]
                distance += abs(current_position[0] - goal_position[0]) +
abs(current_position[1] - goal_position[1])
    return distance
```

Táto funkcia vypočítava Manhattanskú vzdialenosť medzi dvoma stavmi reprezentujúcimi aktuálny stav hlavolamu a jeho cieľový stav. Pre každé nenulové políčko hlavolamu určuje, ako ďaleko je od svojej cieľovej pozície v horizontálnom a vertikálnom smere, a sčíta tieto vzdialenosti pre všetky políčka. Celkový výsledok predstavuje odhad minimálneho počtu ťahov potrebných na dosiahnutie cieľového stavu.

Funkcia pre vykonanie obojsmerného hľadania

Táto funkcia je asi najhlavnejšou časťou môjho programu – vykonáva celkové obojsmerné hľadanie za pomoci ďalších jednoduchších funkcii už spomínaných v tejto dokumentácii. Jej fungovanie opíšem po častiach.

1. Inicializácia

```
start_node = Node(tuple(map(tuple, start_state)))
end_node = Node(tuple(map(tuple, end_state)))

start_queue = [start_node]
end_stack = [end_node]

start_visited = {}
end_visited = {}
```

```
start_visited[start_node.state] = start_node
end_visited[end_node.state] = end_node
```

Táto časť kódu inicializuje začiatočný a koncový uzol reprezentujúci stav hlavolamu. Používam frontu - start_queue a zásobník- end_stack na ukladanie uzlov v procese vyhľadávania od začiatku a konca. Sety start _visited a end_visited používam na uchovávanie navštívených stavov pre rýchlejšie vyhľadávanie a prístup. Začiatočný a koncový uzol sa pridávajú do svojich zoznamov a priradzujú sa im ich stavy.

Ďalej nasleduje while cyklus, v ktorom sa vykonáva hľadanie v oboch smeroch.

2. Hľadanie z počiatočného stavu

```
start_queue.sort(key=lambda node: node.cost +
manhattan_distance(node.state, end_state))
current_node_start = start_queue.pop(0)
current_state_start = current_node_start.state

for move in ['hore', 'dole', 'vlavo', 'vpravo']:
    if valid_move_check(list(map(list, current_state_start)), move):
        new_state_start = apply_move(list(map(list, current_state_start)),
move)
        if tuple(map(tuple, new_state_start)) not in start_visited:
            child_node_start = Node(tuple(map(tuple, new_state_start)),
parent=current_node_start)
            child_node_start.set_move(move)
            child_node_start.set_cost(current_node_start.cost + 1)
            start_queue.append(child_node_start)
            start_visited[tuple(map(tuple, new_state_start))] =
child_node_start

# Check pre pretnutie s koncovým stavom (intersection)
if current_state_start in end_visited:
    intersection_node = end_visited[current_state_start]
    forward_moves = reconstruct_path(current_node_start)
    backward_moves = reconstruct_path(intersection_node, is_reverse=True)
    return forward_moves + backward_moves
```

Táto časť je zodpovedná za vyhľadávanie od počiatočného uzla. Začína sa zoradením start_queue uzlov na základe ich ceny, ktorá kombinuje cenu na dosiahnutie aktuálneho stavu z počiatočného stavu a manhattanskú heuristiku vzdialenosti do koncového stavu. Potom vyberie uzol s najnižšou cenou ako current_node_start. Pre každý zo štyroch možných ťahov ("hore", "dole", "vlavo", "vpravo") skontroluje, či je tento ťah platný v aktuálnom stave pomocou valid_move_check. Ak je pohyb platný, vytvorí sa nový stav použitím pohybu pomocou apply_move a vytvorí sa podriadený uzol, ktorý bude reprezentovať tento nový stav. Cena podriadeného uzla sa aktualizuje a pridá sa do start_queue na ďalšie skúmanie. Potom sa skontroluje, či sa aktuálny stav pretína s koncovým stavom. Ak takýto priesečník nastane, znamená to, že bolo nájdené riešenie. Potom zrekonštruuje priamu a spätnú cestu z počiatočného uzla do uzla priesečníka, resp. z uzla priesečníka do koncového uzla a vráti kombinovanú cestu ako riešenie.

3. Hľadanie z koncového stavu

```
end_stack.sort(key=lambda node: node.cost + manhattan_distance(node.state,
start_state))
current_node_end = end_stack.pop()
current_state_end = current_node_end.state

for move in ['hore', 'dole', 'vlavo', 'vpravo']:
```

```
    if valid_move_check(list(map(list, current_state_end)), move):
        new_state_end = apply_move(list(map(list, current_state_end)),
move)
        if tuple(map(tuple, new_state_end)) not in end_visited:
            child_node_end = Node(tuple(map(tuple, new_state_end)),
parent=current_node_end)
            child_node_end.set_move(move)
            child_node_end.set_cost(current_node_end.cost + 1)
            end_stack.append(child_node_end)
            end_visited[tuple(map(tuple, new_state_end))] = child_node_end

# Check pre pretnutie so začiatočným stavom (intersection)
if current_state_end in start_visited:
    intersection_node = start_visited[current_state_end]
    forward_moves = reconstruct_path(intersection_node)
    backward_moves = reconstruct_path(current_node_end, is_reverse=True)
    return forward_moves + backward_moves
```

Táto časť funguje úplne rovnako ako tá predošlá, s tým rozdielom, že sa vykonáva hľadanie smerom od konca a teda využívajú sa premenné na to určené.

4. Funkcie na parsovanie testovacích súborov a výpisy

Keďže dôležitou súčasťou riešenia je aj testovanie algoritmu, môj program obsahuje funkcie na spracovanie týchto súborov - parse_input_file a parse_state.

parse_input_file: táto funkcia spracováva vstupné údaje zo súboru zadaného pomocou file_path. Zo súboru získa informácie o veľkosti hlavolamu, počiatočnom a konečnom stave, prevedie ich do príslušných dátových štruktúr a vráti tieto hodnoty ako tuple.

parse_state: táto funkcia berie veľkosť hlavolamu a reťazec hodnôt oddelených medzerou, ktoré predstavujú stav, a konvertuje ich na 2D list reprezentujúci stav hlavolamu. Funkcia vráti stav ako 2D list.

V neposlednom rade kód obsahuje funkciu **print_solution_sequence**, ktorá vypíše konečnú postupnosť krokov spolu s vizualizáciou stavov.

Testovanie

Môj program som primárne testovala na rozmeroch hlavolamu 3x3. V maine sa nachádza jednoduché GUI, ktoré sprevádza užívateľa testom/spustením hľadania. Ako prvé sa opýta, či chce riešiť 3x3 alebo 4x4 hlavolam. Po tomto sa spýta, či chce vidieť náhodný hlavolam z testovacej sady, alebo zadať vlastný počiatočný a koncový stav.

Testovacia sada pozostáva z: troch bežných 3x3 vstupov na cca 10 operácií, 1 ťažší 3x3 vstup, edge case kde sa počiatočný stav rovná koncovému a dvoch súborov kde sa testuje hlavolam obrátene, a potom obsahuje 3 4x4 bežné vstupy a jeden ťažší.

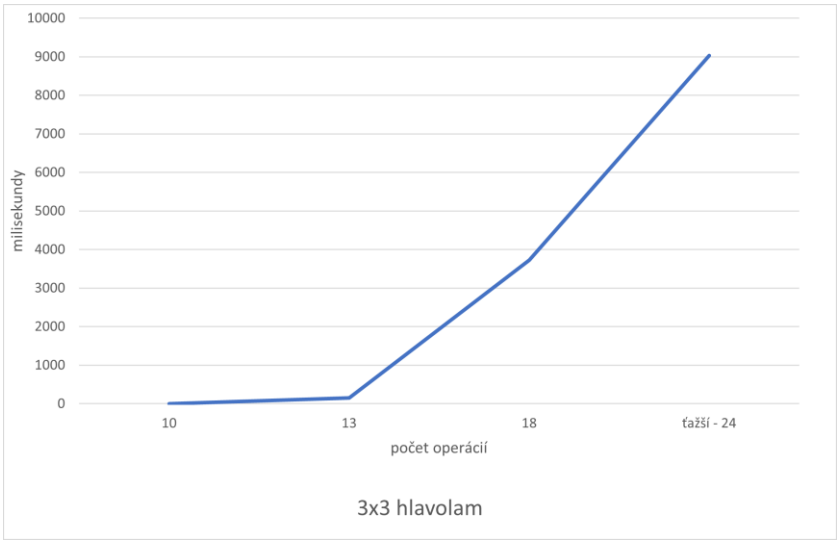
Štruktúra testovacieho súboru je nasledovná:

```
size: 3x3
start: 1 8 2 4 0 5 7 3 6
end: 1 2 3 4 5 6 7 8 0
```

Správnosť implementácie som overila manuálne krokovaním hlavolamu.

Petra Miková
ID: 120852

Výsledky čo sa týka času pre 3x3 hlavolam:



Čo sa týka otestovania prípadu že **zameníme počiatočný stav s koncovým**, použila som tieto stavy:

[[1, 8, 2,], [4, 0, 5], [7, 3, 6]] a [[1, 2, 3], [4, 5, 6,], [7, 8, 0]]

a aj keď som pri oboch príkladoch dostala rovnaký počet operácií pre konečné riešenie, výsledky sa od seba časovo líšili o cca 20 milisekúnd.

V prípade že sa **počiatočný stav rovná koncovému**, algoritmus správne nevypíše žiadnu postupnosť krokov, keďže neexistuje.

Výsledky čo sa týka času pre 4x4 hlavolam:

Petra Miková
ID: 120852

Zhodnotenie

Mojou úlohou bolo implementovať algoritmus obojsmerného hľadania v 8 hlavolame, a túto úlohu sa mi aj podarilo splniť. Program úspešne vyhľadá v rámci jeho možností optimálnu cestu a vráti správnu postupnosť krokov, ktorou sa dostaneme k cieľovému stavu. Implementácia by sa určite dala optimalizovať vyskúšaním napr. viacerých heuristík a ich kombinácií, ale keďže v mojom zadaní podmienka porovnávať heuristiky nebola, rozhodla som sa použiť len heuristiku Manhattanskej vzdialenosti v kombinácii s cenou každého uzla.

Testovanie algoritmu som vykonala na niekoľko rôznych prípadoch – bežných, ťažších, a zopár edge case-och. Pri oboch rozmeroch sa čas zvyšoval so zvyšujúcim sa počtom potrebných operácií. Čo ma možno prekvapilo bol fakt, že pri testovaní dvoch vstupov pri zamenení ich poradia jeden zbehol rýchlejšie ako druhý, aj keď oba pozostávali z rovnakého počtu potrebných operácií. Implementácia je spravená tak, že používateľ vie otestovať aj svoje vlastné vstupy, teda nie je limitovaná len na sadu testovacích príkladov.

Zadanie by som zhodnotila ako veľmi zaujímavé, keďže sa hlavne zo začiatku človek potreboval zamýšľat ako celkovú implementáciu zrealizuje a brať do úvahy viacero faktorov. Zo začiatku som sa napríklad držala striktne len 3x3 rozmeru, neskôr to pozmenila na všeobecnejšie n x n, a verím, že ak by bolo času viac, implementácia by sa dala napasovať aj na rozmery n x m. Testovala som rozmery n = 3 a 4, pretože už pri rozmere 4x4 trocha väčšie vstupy zbehávali dlho, a pri väčšom rozmere by som už nevedela otestovať krokovaním správnosť outputu.