Fakulta informatiky a informačných technológií
Slovenskej technickej univerzity

UMELÁ INTELIGENCIA

Zadanie č.2

8-hlavolam

Lačné hľadanie

Vypracovali:
Matej Voľanský xvolansky@stuba.sk

Matej Voľanský
AIS ID: 103 180

Pondelok 10:00
Cvičiaci: Ing. Juraj Vincúr

Obsah

Matej Voľanský
AIS ID: 103 180

Pondelok 10:00
Cvičiaci: Ing. Juraj Vincúr

Zadanie

Úlohou nášho zadania je nájsť riešenie pre 8-hlavolam, v ktorom sa nachádza 8 očíslovaných políčok od 1 po 8 a jednej medzery. Hlavolam začína vždy s náhodne pomiešanými poličkami a úlohou je posúvať políčka do smeru medzery až do kým neprídeme do cieľa. Posun môže byť o jedno políčko po horizontálnej osi a vertikálnej osi, nie diagonálnej. Teda musíme nájsť správnu postupnosť posunutí. Napríklad môže byť vygenerovaná následovná začiatočná a koncová pozícia (0 reprezentuje medzeru):

```
Start:              End:
[6, 5, 0]            [6, 1, 0]
[3, 2, 8]            [4, 3, 8]
[7, 4, 1]            [2, 7, 5]
```

Riešenie

Algoritmus lačného hľadania (Greedy algorithm)

Ako aj z názvu vyplýva, tento algoritmus je “pažravý” alebo “lačný” čo v pre nás znamená že vždy si vybere cestu s najnižšou cenou. Výhodou tohto algoritmu je, že pre problémy menších rozmerov je celkom priamočiary a jednoduchý. Nevýhodou je, že najlepšie krátkodobé rozhodnutia môžu viesť k najhorším dlhodobým riešením v budúcnosti.

Tento algoritmus k svojej činnosti potrebuje dodatočné informácie o riešenom probléme, takzvané heuristické funkcie. Pre naše riešenie používame nasledovné:

1. Počet políčok, ktoré nie sú na svojom mieste
2. Súčet vzdialeností jednotlivých políčok od ich cieľovej pozície
3. Kombinácia predchádzajúcich odhadov

```
def greedy_solve(puzzle, heuristic):
    counter, act, visited = 1, Node(puzzle), list()
    choices = queue.PriorityQueue()
    act.value = act.get_cost(heuristic)
    choices.put((act.value, counter, act))
    timeout = time.time() + 15
```

Na začiatok si algoritmus zadefinuje potrebné premenné:

counter - koľko uzlov bolo vytvorených
act - aktuálny uzol
visited - pole už navštívených tabuliek 8 hlavolamu
choices - zásobník možností pre ďalší krok, do neho sa ukladá pole s hodnotou heuristiky uzlov, poradové číslo uzla a daný uzol
timeout - časovač pre neriešiteľné hlavolamy

```
while time.time() < timeout:
    if choices.empty(): break
    act = choices.get()[2]
    if act.is_finished():
        act.state.puzzle_print("Result")
        print("Steps: ", str(len(act.moves)), "Moves: ", act.moves)
        return True
```

Následne začne hľadať riešenia pokiaľ mu nevyprší čas. Časovač je nastavený na 15 sekúnd. Na začiatok skontroluje, či sa mu zásobník možností už nevyčerpal. V takom prípade sa jedná o hlavolam, ktorý nemá riešenie. Ak pole nie je prázdne, vytiahne zo zásobníka prvú najlepšiu možnosť. Na začiatku bol do zásobníka vložený štartovný uzol, takže prve vytiahne ten. Potom skontroluje či aktuálny uzol nie je koncový uzol riešenia. Ak áno tak nam vypíše informácie o riešení ako koľko krokov a aké kroky boli potrebné na vyriešenie hlavolamu.

```
    elif act.state.deck not in visited:
        # print(counter, act.value, act.state.deck, act.moves)
        visited.append(act.state.deck)
        children = act.create_children_nodes()
        while not children.empty():
            counter += 1
            child = children.get()
            child.value = child.get_cost(heuristic)
            choices.put((child.value, counter, child))
print("Solution wasn't found")
return False
```

Ak aktuálny uzol nie je koncový uzol, pridá tabuľku aktuálneho uzlu už do navštívených uzlov aby sa algoritmus necyklil medzi rovnakými pohybmi. Následne sa zavolá funkcia create_chilrdren_nodes(), ktorá nám vráti zásobník so všetkými novými uzlami s možnými pohybmi. Teda ak je medzera v strede tabuľky, tak v zásobníku budu štyri uzly pre pohyb hore, dole, do ľava, do prava. Ak medzera bude v ľavom hornom kraji, tak v ňom budú uzli s pohybmi dole a do prava.

Následne cyklíme celým zásobníkom kym sa nevyčerpá. V cykle pre každý nový uzol pripočítame **counter**. Pre každý nový uzol (potomok aktuálneho uzla - child) sa vypočíta hodnota danej heuristiky a vloží sa do zásobníka možností. Teda ak vonkajší cyklus znova iteruje, aktuálny uzol ktorý vyberie zo zásobníka možností bude pohyb s najmenšou hodnotou heuristiky. Takto pokračuje kým nenájde riešenie.

Môže nastať situácia, že v zásobníku bude x nových možnosti s rovnakou hodnotou heuristiky. Preto je do zásobníka vložené poradové číslo, vďaka ktorému vyberie prvý vložený pohyb s rovnakou heuristikou ako ostatné. Inak by zásobník vrátil náhodný uzol.

Vieme povedať že algoritmus používa DFS (Depth-first search) teda hľadá riešenia do hĺbky.

Aby zásobník vedel porovnávať hodnoty uzlov, museli sme mu vytvoriť funkcie podľa ktorých vie ako má porovnávať hodnoty.

```
def __lt__(self, other):
    return self.value < other.value

def __gt__(self, other):
    return self.value > other.value

def __le__(self, other):
    return self.value < other.value or self.value == other.value

def __ge__(self, other):
    return self.value > other.value or self.value == other.value
```

Algoritmus berie ako parametre daný hlavolam a číslo heuristiky. Heuristika sa vyberá pomocou funkcie get_cost(heuristic). Tretia heuristika je kombinácia dvoch predošlých.

```
def get_cost(self, heuristic):
    if heuristic == 1: return self.heuristic1()
    if heuristic == 2: return self.heuristic2()
    if heuristic == 3: return self.heuristic1() + self.heuristic2()
```

Prvá heuristika je počet nesprávne položených políčok. Ráta sa jednoduchým iterovaním celou tabuľkou a porovnávaním hodnot na políčkach aktuálnej tabuľky s výslednou tabuľkou.

```
def heuristic1(self):
    sum_same = 0
    for ix, row in enumerate(self.state.deck):
        for iy, value in enumerate(row):
            if not value == self.state.goal[ix][iy]: sum_same += 1
    return sum_same
```

Druhá heuristika je súčet všetkých vzdialeností jednotlivých políčok od ich cieľovej pozície. Vytvoria sa dve knižnice pre aktuálne a cieľové pozície. Teda ako kľúč je použité číslo na danom políčku a ako hodnota kľúča sú súradnice daného čísla. Následne sa obe knižnice potredia aby čísla išli od 1 (0, teda medzera sa ignoruje) po koncové číslo. Tak sa pre každé číslo získa vzdialenosť absolútnou hodnotou odčítania aktuálnej pozície od cieľovej, ktoré sa sumuje do celkovej hodnoty **total_sum**.

```
def heuristic2(self):
    xy_start, xy_end, total_sum = {}, {}, 0
    # For the actual deck and the end deck, fill dictionary with deck value as key and position as dict value
    for ix, row in enumerate(self.state.deck):
        for iy, value in enumerate(row):
            if not value == 0: xy_start[value] = [ix, iy]
    for ix, row in enumerate(self.state.goal):
        for iy, value in enumerate(row):
            if not value == 0: xy_end[value] = [ix, iy]
    # Sort both dictionaries so we can iterate through them and compare
    xy_start, xy_end = dict(sorted(xy_start.items())), dict(sorted(xy_end.items()))
    for i in range(1, self.state.x * self.state.y):
        total_sum += abs(xy_start[i][0] - xy_end[i][0]) + abs(xy_start[i][1] - xy_end[i][1])
    return total_sum
```

Matej Voľanský
AIS ID: 103 180

Reprezentácia údajov

Tabuľka hlavolamu je reprezentovaná dvojrozmerným poľom: [[0, 1, 2], [3, 4, 5,], [6, 7, 8]]

Na prácu a pohyb s tabuľkami je použitá trieda Puzzle, do ktorej sa zadáva rozmer tabuľky, aktuálna a cieľová tabuľka. V triede je knižnica so všetkými možnými pohybmi ako kľúč a ako hodnota je funkcia pre daný pohyb.

```
class Puzzle:
    def __init__(self, x, y, deck=None, goal=None):
        self.deck, self.goal = deck, goal
        self.x, self.y = x, y
        if deck is not None: self.gap = self.get_gap_pos()
        self.moves = {"U": self.up, "D": self.down, "L": self.left, "R": self.right}
```

Na manuálne nastavenie tabuliek slúži funkcia set_deck_goal a funkcia get_gap_pos slúži na nájdenie pozície medzery keďže v našom programe pohyby sa vykonávajú z pohľadu medzery.

```
def set_deck_goal(self, deck, goal):
    self.deck, self.goal = deck, goal
    self.gap = self.get_gap_pos()

def get_gap_pos(self):
    for ix, row in enumerate(self.deck):
        for iy, value in enumerate(row):
            if value == 0: return [ix, iy]
```

Pre každý pohyb máme osobitnú funkciu, ktorá vracia novú tabuľku už s vykonaným pohybom.

```
def up(self):
    if not self.gap[0] == 0: return self.swap(self.gap[0] - 1, self.gap[1])
    else: return [row[:] for row in self.deck]

def down(self):
    if not self.gap[0] == self.x - 1: return self.swap(self.gap[0] + 1, self.gap[1])
    else: return [row[:] for row in self.deck]

def left(self):
    if not self.gap[1] == 0: return self.swap(self.gap[0], self.gap[1] - 1)
    else: return [row[:] for row in self.deck]

def right(self):
    if not self.gap[1] == self.y - 1: return self.swap(self.gap[0], self.gap[1] + 1)
    else: return [row[:] for row in self.deck]

def move(self, move):
    if move in self.moves: return self.moves[move]()
```

Výmena dvoch políčok je realizovaná funkciou swap, ktorá z aktuálnej tabuľky vytvorí novú a v nej vykoná zadaný pohyb výmenou políčok.

```
def swap(self, x1, y1):
    new_list = [row[:] for row in self.deck]
    temp = new_list[self.gap[0]][self.gap[1]]
    new_list[self.gap[0]][self.gap[1]] = new_list[x1][y1]
    new_list[x1][y1] = temp
    return new_list
```

Jednotlivé uzly reprezentujeme triedou Node. Pri vytvorení dostane aktuálnu tabuľku, a ak jeho rodič existuje, tak aj všetky pohyby rodiča plus pohyb tohto uzla. Taktiež v sebe ukladá hodnotu heuristiky.

```
class Node:
    def __init__(self, deck, move="", parent=None):
        self.state = deck
        if parent is None: self.moves = move
        else: self.moves = parent.moves + move
        self.value = 0
```

Dôležitou funkciou tejto triedy je create_children_nodes().

```
def create_children_nodes(self):
    # Create queue of children nodes for all possible moves
    children = queue.Queue()
    for move in self.state.moves:
        new_deck = Puzzle(self.state.x, self.state.y, self.state.move(move), self.state.goal)
        if not new_deck.gap[0] == self.state.gap[0] or not new_deck.gap[1] == self.state.gap[1]:
            children.put(Node(new_deck, move, self))
    return children
```

Na začiatku si vytvorí nový zásobník. Pre každý možný pohyb sa vytvorí nová inštancia triedy Puzzle s názvom **new_deck** do ktorej sa vloží tabuľka už s vykonaným pohybom. Tak sa skontroluje či sa pohyb vykonal, teda medzera novej tabuľky nie je na rovnakej pozícii ako v starej. Ak nie je tak do zásobníka sa vloží nový uzol s touto tabuľkou a pohybom. Funkcia vráti tento zásobník, ktorý sa potom vkladá do zásobníka v algoritme a z neho následne sa vyberajú možnosti.

Matej Voľanský
AIS ID: 103 180

Pondelok 10:00
Cvičiaci: Ing. Juraj Vincúr

Testovanie

Pri testovaní nášho riešenia nás zaujíma celkový čas nájdenia riešenia a počet vykonaných krokov.
Náš tester vyzerá následovne:

```
def tester(min_size, max_size):
    i = min_size
    while i <= max_size:
        j = min_size
        while j <= max_size:
```

Ako parameter dostane najmenšiu veľkosť a najväčšiu veľkosť tabuľky. Potom idú dva cykly, pre ktoré sa vlastne testujú M x N vygenerované tabuľky.

```
start_deck, end_deck = deck_generator(i, j), deck_generator(i, j)
start_timer = time.time()
# p = Puzzle(2, 5, [[1, 0, 2, 3, 4], [5, 6, 7, 8, 9]], [[5, 1, 9, 8, 7], [2, 4, 3, 6, 0]])
p = Puzzle(i, j, start_deck, end_deck)
print("Heuristic 1\n")
p.puzzle_print("Start")
if greedy_solve(p, 1) is not False:
    print("Time elapsed: " + str(time.time() - start_timer) + "\nHeuristic 2\n")
    start_timer = time.time()
    greedy_solve(p, 2)
    print("Time elapsed: " + str(time.time() - start_timer) + "\nHeuristic 3\n")
    start_timer = time.time()
    greedy_solve(p, 3)
    print("Time elapsed: " + str(time.time() - start_timer))
```

Pre každú heuristiku sa spustí vlastný časovač. Vygeneruje sa začiatočná a cieľová tabuľka. Pre neriešiteľnosť tabuľky máme podmienku že ak prvé spustenie algoritmu s prvou heuristikou nenašlo riešenie, teda hľadanie trvalo dlhšie ako 15 sekúnd, vytvoria sa nové tabuľky a toto sa opakuje pokiaľ algoritmus nevyrieši hlavolam. Ak ho vyrieši tak sa spustia algoritmy s ďalšími heuristikami a vypíšu sa časy a kroky.

Na generovanie tabuliek máme funkciu:

```
def deck_generator(x, y):
    choices = random.sample(list(range(0, x * y)), k=x * y)  # Randomly arranged list [1,7,3,8,4,5]
    return [choices[i:i + y] for i in range(0, x * y, y)]  # Make sub lists [[1,7,3],[8,4,5]]
```

Prve sa vytvorí pole hodnot od 0 po M x N a následne sa hodnoty v ňom náhodne poprehadzujú. Tak sa vráti pole porozdelované na riadky o šírke stĺpca.

Výsledky

Program vygeneroval nasledovné tabuľky.

```
Start:          End:
[1, 2]          [0, 3]
[0, 3]          [2, 1]
```

```
Start:             End:
[4, 2, 0, 3]       [5, 2, 7, 0]
[5, 6, 7, 1]       [3, 6, 1, 4]
```

```
Start:          End:
[1, 4, 0]       [0, 3, 4]
[3, 2, 5]       [1, 5, 2]
```

Matej Voľanský
AIS ID: 103 180

Pondelok 10:00
Cvičiaci: Ing. Juraj Vincúr

```
Start:          End:
[5, 4]          [3, 1]
[1, 2]          [2, 5]
[0, 3]          [0, 4]
```

```
Start:              End:
[10, 1, 3, 9]           [11, 4, 6, 2]
[5, 0, 4, 7]            [0, 9, 3, 8]
[2, 8, 6, 11]           [10, 7, 1, 5]
```

```
Start:          End:
[6, 5, 0]        [6, 1, 0]
[3, 2, 8]        [4, 3, 8]
[7, 4, 1]        [2, 7, 5]
```

```
Start:          End:
[1, 2]          [3, 5]
[7, 0]          [6, 2]
[6, 3]          [0, 7]
[4, 5]          [4, 1]
```

```
Start:          End:
[5, 7, 6]        [3, 5, 2]
[8, 0, 2]        [8, 0, 4]
[9, 4, 10]       [10, 11, 9]
[1, 3, 11]       [1, 7, 6]
```

```
Start:              End:
[10, 5, 13, 15]             [2, 6, 9, 5]
[2, 4, 0, 7]            [4, 7, 8, 0]
[8, 14, 1, 3]           [11, 1, 3, 15]
[9, 12, 6, 11]          [14, 10, 12, 13]
```

Počet krokov

H / M x N	2x2	2x3	2x4	3x2	3x3	3x4	4x2	4x3	4x4
H 1	5	18	51	20	48	109	56	64	173
H 2	5	18	45	12	66	91	74	76	157
H 3	5	18	51	12	48	75	70	106	129

Čas v ms

H / M x N	2x2	2x3	2x4	3x2	3x3	3x4	4x2	4x3	4x4
H 1	0.4	9.3	149	10.2	70	893	176	256	2732
H 2	0.4	2.8	27	1.07	87	218	101	131	776
H 3	0.4	3.9	25	1.1	47	38	78	264	281

Matej Voľanský
AIS ID: 103 180

Pondelok 10:00
Cvičiaci: Ing. Juraj Vincúr

Zhodnotenie

Naše riešenie je stále možné vylepšiť či už pamätovo alebo rýchlosťou. Napríklad pri výpočte heuristiky 2 by sme mohli **xy_end** mať vypočítané iba raz a nepočítať ho odznova pre každý uzol. Taktiež v uzloch by sme nemuseli ukladať cieľovú tabuľku. Napriek výhradám, naše riešenie funguje pre každú riešiteľnú tabuľku. Bohužiaľ riešiteľnosť tabuľky sme v našom programe neimplementovali z dôvodu možnosti riešenia aj M x N tabuliek. V prvej verzii bola takáto funkcia implementovaná pomocou počtom inverzií mod 2. Následne sme zistili, že to platí iba pre NxN tabuľky a preto sme opustili od tejto funkcie. Riešiteľnosť je jednoducho daná 15 sekundovým časovačom pre algoritmus. Pre väčšie tabuľky ako 4x4 to môže byť ale málo a bolo by potrebné to manuálne prenastaviť v programe na väčší čas.

Výsledky testovania sú zaujímavé, môžeme vidieť preteky o lepší čas medzi heuristikou 2 a heuristikou 3. Heuristika 1 je zo všetkých heuristík najpomalšia a priemerne vyžadovala najviac krokov na nájdenie riešenia. Samozrejme záleží aj náročnosť vygenerovanej tabuľky, ktorú nevieme ovplyvniť.

Ak si dáme v cyklu algoritmu vypísať počet uzlov, hodnotu heuristiky a aktuálnu tabuľku plus vykonané pohyby, tak napríklad s heuristikou 3 pre takúto vstup môžeme vidieť takýto rozvoj

```
Start:          End:
[4, 0]          [4, 5]
[1, 3]          [1, 3]
[5, 2]          [2, 0]
```

```
Heuristic 3

1 7 [[4, 0], [1, 3], [5, 2]]
3 9 [[4, 3], [1, 0], [5, 2]] D
6 9 [[0, 4], [1, 3], [5, 2]] L
8 9 [[4, 3], [1, 2], [5, 0]] DD
10 9 [[4, 3], [1, 2], [0, 5]] DDL
12 11 [[4, 3], [0, 1], [5, 2]] DL
15 10 [[4, 3], [5, 1], [0, 2]] DLD
17 7 [[4, 3], [5, 1], [2, 0]] DLDR
19 9 [[4, 3], [5, 0], [2, 1]] DLDRU
22 7 [[4, 0], [5, 3], [2, 1]] DLDRUU
24 8 [[4, 3], [0, 5], [2, 1]] DLDRUL
27 9 [[0, 4], [5, 3], [2, 1]] DLDRUUL
29 8 [[5, 4], [0, 3], [2, 1]] DLDRUULD
32 10 [[0, 3], [4, 5], [2, 1]] DLDRULU
34 10 [[4, 3], [2, 5], [0, 1]] DLDRULD
36 8 [[4, 3], [2, 5], [1, 0]] DLDRULDR
38 10 [[5, 4], [2, 3], [0, 1]] DLDRUULDD
40 8 [[5, 4], [2, 3], [1, 0]] DLDRUULDDR
42 10 [[5, 4], [3, 0], [2, 1]] DLDRUULDR
45 8 [[5, 4], [3, 1], [2, 0]] DLDRUULDRD
47 10 [[4, 3], [2, 0], [1, 5]] DLDRULDRU
50 8 [[4, 0], [2, 3], [1, 5]] DLDRULDRUU
52 10 [[0, 4], [2, 3], [1, 5]] DLDRULDRUUL
54 11 [[1, 4], [0, 3], [5, 2]] LD
57 10 [[1, 4], [5, 3], [0, 2]] LDD
59 7 [[1, 4], [5, 3], [2, 0]] LDDR
61 10 [[1, 4], [5, 0], [2, 3]] LDDRU
64 9 [[1, 4], [0, 5], [2, 3]] LDDRUL
67 7 [[0, 4], [1, 5], [2, 3]] LDDRULU
69 5 [[4, 0], [1, 5], [2, 3]] LDDRULUR
71 3 [[4, 5], [1, 0], [2, 3]] LDDRULURD
Result:            End:
[4, 5]          [4, 5]
[1, 3]          [1, 3]
[2, 0]          [2, 0]
Steps:  10 Moves:  LDDRULURDD
Time elapsed: 0.0031347274780273438
```

Matej Voľanský
AIS ID: 103 180

Pondelok 10:00
Cvičiaci: Ing. Juraj Vincúr

Používateľská príručka

Na zhotovenie nášho riešenie sme použili tieto knižnice.
Na testovanie stačí napísať do kódu na poslednom riadku hodnoty testera ako parametre.

```
tester(2, 3)
```