

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií
23. Október 2021

Umelá inteligencia
Riešenie 8-hlavlámu
A* algoritmus

Roland Vdovják
id: 110912

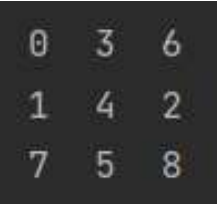
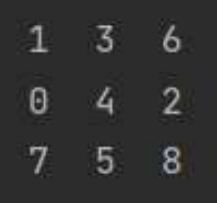
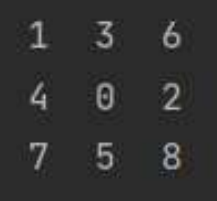
Riešenie

Pri riešení sme postupovali nasledovne. Najskôr sme vytvorili maticu (dvojrozmerné pole), ktorá reprezentuje stav hlavolamu. Tento krok retrospektívne nebol až tak vhodný, lebo reprezentácia maticou spomalila značne program. Podľa vstupnej matice zisťuje program veľkosť ($m \times n$, m je hodnota na osi x a n je hodnota na osi y). Ďalej sme vytvorili niekoľko jednotlivých funkcií na výpis matice, obidve heuristiky a vytvorili objekt uzla „*NODE*“.

Zisťovanie možného pohybu

Keď sme mali pripravené tieto pomocné funkcie, začali sme pracovať na pohybe stavov a algoritme. Na to, aby sme vedeli, ako môžeme posúvať políčka na voľné miesto, musíme zistiť, kde sa nachádza medzera a z toho vyvodiť, aké posuvy sú možné.

Môžu nastať nasledujúce prípady:

	Voľné miesto (nula „0“) sa nachádza v rohu hlavolamu. Možné sú len dva pohyby- doľava, hore.
	Voľné miesto (nula „0“) sa nachádza na kraji hlavolamu. Možné sú len tri pohyby- dole, doľava, hore.
	Voľné miesto (nula „0“) sa nachádza nie na kraji a nie v rohu (pre prípad 3x3 v strede). Možné sú všetky pohyby.

Riešenie týchto situácií máme pomocou porovnávania pozície nuly s rozmermi poľa. Každý z pohybov má svoje číslo, mocninu desiatky, tieto čísla sa sčítajú a obrátia, **výsledkom je string tvaru „1101“** čo znamená, že je možný pohyb dole(1), hore(10) a pohyb doprava(1000). Od tohto čísla **sa odčítava predchádzajúci pohyb**, resp. pohyb nadradeného uzla. Tým pádom sa mocnina desiatky s daným pohybom odčíta.

Ukážka kódu:

```
def check_move(input):
    num = 0
    #UP
    if input.space[0] != str(n - 1):
        num += 1
    #DOWN
    if input.space[0] != '0':
        num += 10
    #LEFT
    if input.space[2] != str(m - 1):
        num += 100
    #RIGHT
    if input.space[2] != '0':
        num += 1000

    num -= input.operation

    return str(num).rjust(4, "0")[:-1]
```

Pohyb

Pri pohybe sa mení miesto nuly (voľné miesto) s miestom z ktorého sa pohybuje. X a Y predstavujú polohu na osi x a na osi y voľného miesta v stave. Pohyb aj funkcia je pomenovaná analogicky podľa pohybu, ktorý je dovolený v stave. Je to preto, aby hlavný algoritmus mohol volať funkcie podľa toho, čo je stavu dovolené.

Výmena hodnôt prebieha bez dočasnej premennej, pretože vieme, že vždy je jedna z hodnôt nula, tú prepíšeme číslom zo smeru, z ktorého ideme a toto číslo nahradíme nulou (zvýraznené žltou). Takto sa mení poloha nuly zo všetkých smerov, v príklade je pohyb „dole“.

Ukážka kódu:

```
def m1(input):
    matrix = copy.deepcopy(input)

    x = int(matrix.space[2])
    y = int(matrix.space[0])
    matrix.matrix[y][x] = matrix.matrix[y + 1][x]
    matrix.matrix[y + 1][x] = 0

    matrix.G = input.G + 1
    matrix.space = "{} {}".format(y+1,x)
    matrix.parent = input
    matrix.operation = 10

    return matrix
```

Algoritmus A*

Algoritmus je reprezentovaný funkciou „a_star()“, ktorá má za argumenty začiatok, teda počiatočný uzol a typ heuristiky.

Algoritmus sa opakuje, dokým heuristika daného stavu nie je nulová alebo je hĺbka hľadania väčšia ako 37 (prípadná neriešiteľnosť alebo veľká náročnosť riešenia, pri implementácii v pythone by zabrala až moc veľa času, tak program končí bez riešenia). Každým opakovaním cyklu sa do momentálneho uzla „current“ priradí uzol z predchádzajúceho opakovania, ak cyklus ide prvýkrát, v uzle sa nachádza začiatkový uzol, resp. stav.

Pri opakovaní sa vždy zisťujú možné pohyby posledne vytvoreného uzla. Po zistení pohybov sa vo for cykle **vykonávajú pohyby**, ktoré sú realizovateľné. Ak tento nový stav už existuje (**zisťujeme v tabuľke(hashtable „Htable“)**), tak pokračujeme na ďalší pohyb. Ak stav ešte neexistuje, je potrebné dokončiť vytváranie uzla a pridať stav do tabuľky.

Po všetkých možných pohyboch sa umiestňujú vhodné uzly do heap queue, ktorá je zoradená podľa minimálnej hodnoty uzla A, čo predstavuje súčet heuristiky a vzdialenosti od začiatku(v prípade rovnosti hodnoty A sa porovnáva samostatná heuristika). Na záver funkcia vráti finálny uzol, ktorý ďalej spracovávame vo funkcii main.

Ukážka kódu:

```
def a_star(start, h_type):
    root = copy.deepcopy(start)
    heapq.heappush(heap, root)
    while root.H != 0:
        current = copy.deepcopy(root)
        move_options = check_move(current)

        for i in range(4):
            if move_options[i] == '1':
                current = eval("m" + str(10**i) + '(current)')

                if Htable.get(str(current.matrix)) is None:
                    Htable[str(current.matrix)] = True
                    current.H = eval("heuristic_" + h_type +
                                     '(current.matrix)')
                    current.A = current.G + current.H
                    q.append(current)
                    current = current.parent

        for i in range(len(q)):
            heapq.heappush(heap, q[i])
        q.clear()
        root = heapq.heappop(heap)

    return root
```

Výhody a nevýhody

Ako sme už stihli spomenúť, jednou z nevýhod je reprezentácia stavu pomocou matice, resp. numpy poľa. Riešenie uvedené na stránke (64bit integer) je jednoznačne efektívnejšie.

S týmto problémom je spojené mnoho ďalších problémov čo sa týka efektívnosti, no zjednodušení to písanie a chápanie programu.

Ďalej za prípadné zlepšenie do budúcnosti považujeme zmenu typov premenných a tým pádom zmenšenie počtu pretypovaní a pracovaní so stringami.

Za výhody považujeme používanie hash mapy na stavy, ktoré už poznáme, vyradovanie spätného pohybu za použitia posledného použitého operátora. Riešenie je taktiež prehľadné, dostatočne okomentované a riadne otestované, viac v časti [Testovanie](#) a [Výsledky](#). Neplytvali sme nadbytočnými údajmi pri vytváraní uzlov. Jedinou diskutabilnou premennou objektu je pamätanie si súčtu heuristiky a doterajšou cestou. Túto premennú však používame na zoradovanie resp. pridávanie do heap queue. Ak sa premenné rovnajú, využívame aj hodnotu heuristiky. Pamätanie premennej so súčtom zmenšujeme náročnosť. Magická metóda Pythonu `__lr__` by obsahovala 2 súčty, pri každej heap queue operácii push by sa muselo sčítavať.

Za ďalší problém, ktorý spomaľuje program považujeme priradovanie uzlov do zoznamu a až po celej úrovni vkladáme do heapq. Toto riešenie dovolilo programu fungovať správne, no zásadne spomalilo ho.

Program je písaný v Pythone, čo nie je optimálne pre časové porovnávanie, keďže je podstatne pomalší ako C/C++. Naším vylepšením je kompilovanie pomocou Cython-u, ktorý program zrýchli. Pre python sme sa rozhodli, pretože sa chceme zlepšiť v tomto jazyku, začali sme s ním nedávno.

Výhodou A* algoritmu je prechádzanie nie všetkých uzlov, napríklad pri našom najhlbšom hľadaní (cesta má hodnotu 31) program spracováva 161 502 a 7 943 uzlov (podľa použitej heuristiky) a nie 2^{31} uzlov.

Testovanie

Program sme testovali na stavoch o rozmeroch 2x3, 3x2, 2x4, 4x2 a 3x3. Každý rozmer sme otestovali piatimi možnými vstupmi. Tieto každý z týchto vstupov sme spustili päťkrát a čas spriemerovali. Na záver sme urobili grafy a zistené dáta popísali. Vstup sme zadávali manuálne zo súboru „test.txt“, keďže nebolo nikde spomínané, že vstupy majú byť načítavané inak. V každom z rozmerov, kde to bolo možné, sme použili aj vstup zo stránky premetu UI.

Vstup 3x2 a 2x3 má rovnakú náročnosť, no testovaním obidvoch rozložení demonštrujeme schopnosť programu riešiť osemhlavolam o $m \times n$ rozmeroch. Koncový stav sme pri testovaní nemenili, až na poslednú úlohu.

Vstupy boli nasledovné:

# VSTUP 3x2	vstup	# VSTUP 2x3	vstup
# Zaciatky		# Zaciatky	
([[3,4,5], [0,1,2]])	1	([[3,0], [4,1], [2,5]])	1
([[5,3,4], [1,2,0]])	2	([[5,3], [4,1], [2,0]])	2
([[4,0,5], [3,1,2]])	3	([[2,3], [4,5], [1,0]])	3
([[2,0,1], [4,5,3]])	4	([[2,5], [4,1], [0,3]])	4
([[3,4,5], [1,2,0]])	5	([[4,5], [0,1], [3,2]])	5
# Koniec		# Koniec	
([[0,1,2], [3,4,5]])		([[0,1], [2,3], [4,5]])	
# VSTUP 4x2		# VSTUP 2x4	
# Zaciatky		# Zaciatky	
([[3,2,5,4], [7,6,1,0]])	1	([[7,3], [5,6], [2,1], [4,0]])	1
([[7,1,5,4], [3,6,2,0]])	2	([[7,3], [6,0], [2,1], [4,5]])	2
([[0,1,5,6], [3,4,2,7]])	3	([[5,4], [6,0], [2,1], [3,7]])	3
([[5,4,2,0], [3,7,6,1]])	4	([[5,6], [0,4], [7,1], [3,2]])	4
([[0,4,2,5], [3,7,6,1]])	5	([[7,0], [6,2], [1,5], [3,4]])	5
# Koniec		# Koniec	
([[0,1,2,3], [4,5,6,7]])		([[0,1], [2,3], [4,5], [6,7]])	
# VSTUP 3x3			
# Zaciatky			
([[0,6,1], [7,5,4], [8,2,3]])	1		
([[2,4,1], [8,0,3], [7,6,5]])	2		
([[4,5,8], [1,0,6], [3,2,7]])	3		
([[1,2,3], [4,5,6], [7,8,0]])	4		
([[8,0,6], [5,4,7], [2,3,1]])	5		
# Koniec			
([[0,1,2], [3,4,5], [6,7,8]])			

Výsledky

Pri všetkých rozmeroch bola heuristika 2 (vzdialenosť políčka od svojho cieľového umiestnenia) efektívnejšia, čo sa týka počtu spracovaných aj prejdenných uzlov. Rozdiel medzi spracovanými a vytvorenými uzlami je taký, že vytvorí len taký uzol, ktorého stav sa ešte nenachádza medzi stavmi vytvorených uzlov.

Časová náročnosť sa preukazovala hlavne na väčšom rozmere hlavolamu, heuristika 2 bola časovo rýchlejšia. Pri malom počte uzlov bola efektívnejšia heuristika 1. Dôvodom tohto je jednoduchší výpočet tejto heuristiky. V ostatných prípadoch zložitejšia heuristika využívala menej uzlov, tým pádom sa stala rýchlejšou.

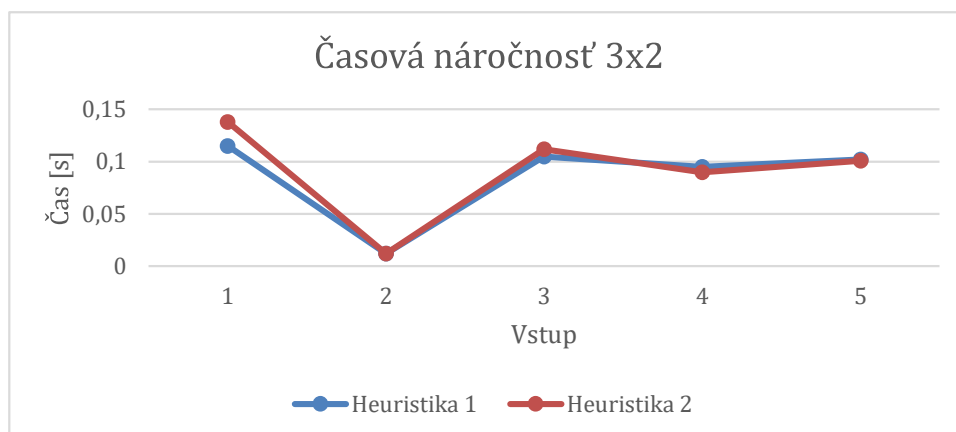
Pre päť rôznych vstupov na každý rozmer sme sa rozhodli preto, aby bolo jasnejšie vidieť závislosť od vstupu a aby sme si vedeli urobiť lepší obraz o rozpätí náročnosti jednotlivých rozmerov, nie všetky vstupy sú rovnako náročné.

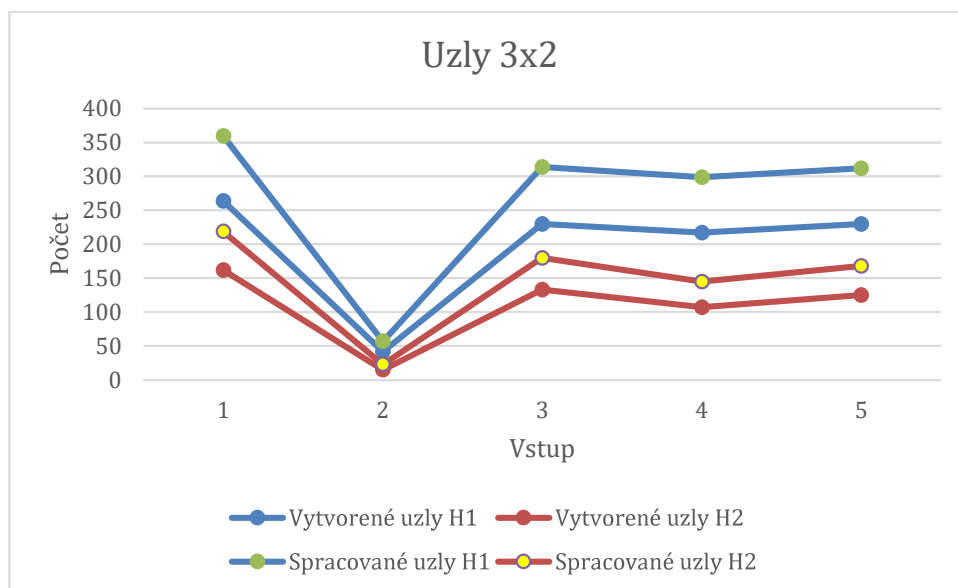
Rozmer 3x2

Najmenší možný rozmer. Pri rozmere 2x2 sa medzera a čísla s ňou točia po obvode. Ak je úloha riešiteľná, tak rozmer 2x2 má najväčšiu možnú zložitost' 3, čo je pre testovanie zanedbateľné.

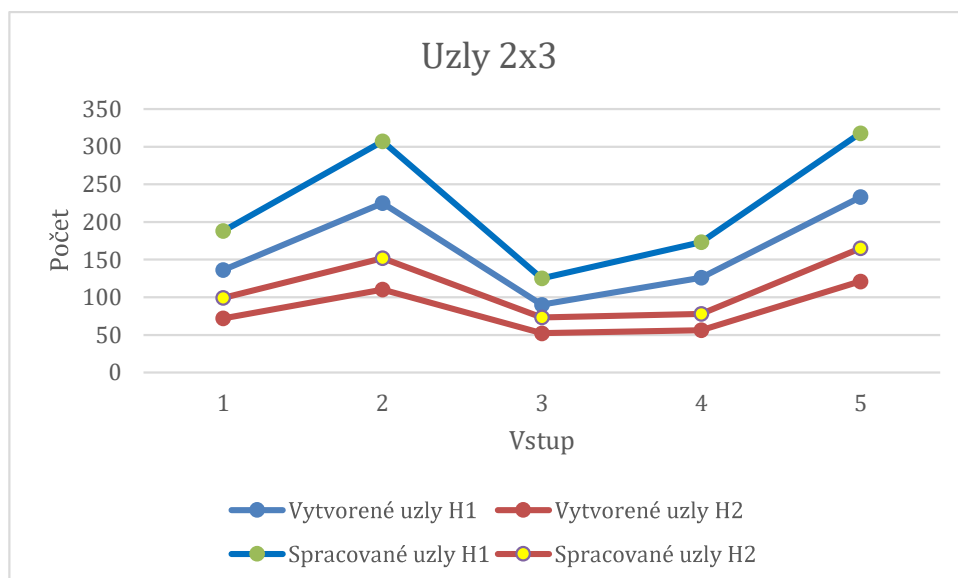
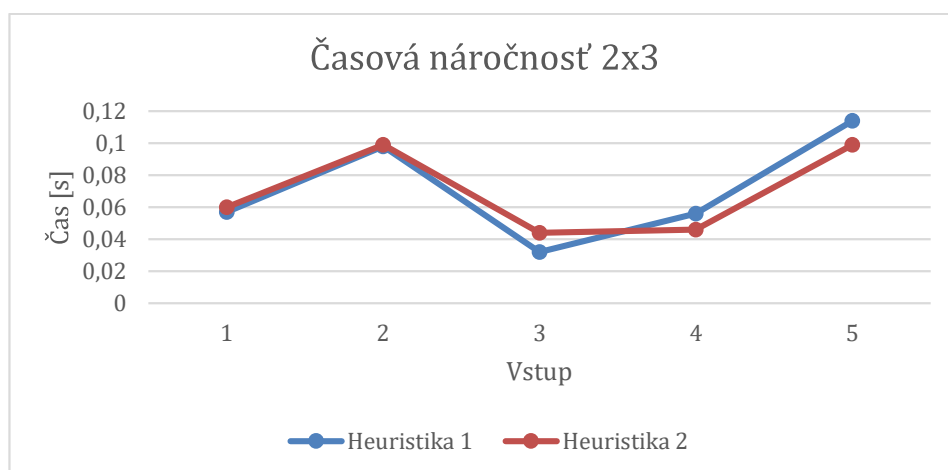
Rozmer 3x2 a jeho otočená interpretácia 2x3 mali vstupy, ktoré boli jedinými prípadmi, pri ktorých bola heuristika 1 časovo efektívnejšia. Počet uzlov sa pohyboval v stovkách (viď tabuľka). Pamäťová efektívnosť bola od začiatku lepšia pri heuristike 2.

	3x2 Uzly				2x3 Uzly				
	H1		H2			H1		H2	
Vstup	Vytvorené	Spracované	Vytvorené	Spracované	Vstup	Vytvorené	Spracované	Vytvorené	Spracované
1	136	188	72	99	1	264	360	162	219
2	225	307	110	152	2	42	57	15	23
3	90	125	52	73	3	230	314	133	180
4	126	173	56	78	4	217	299	107	145
5	233	318	121	165	5	230	312	125	168





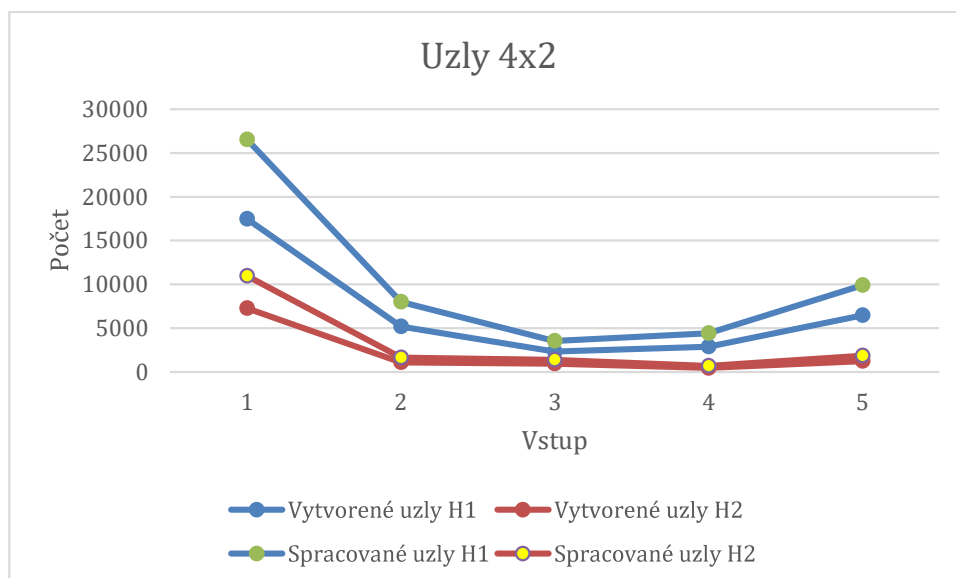
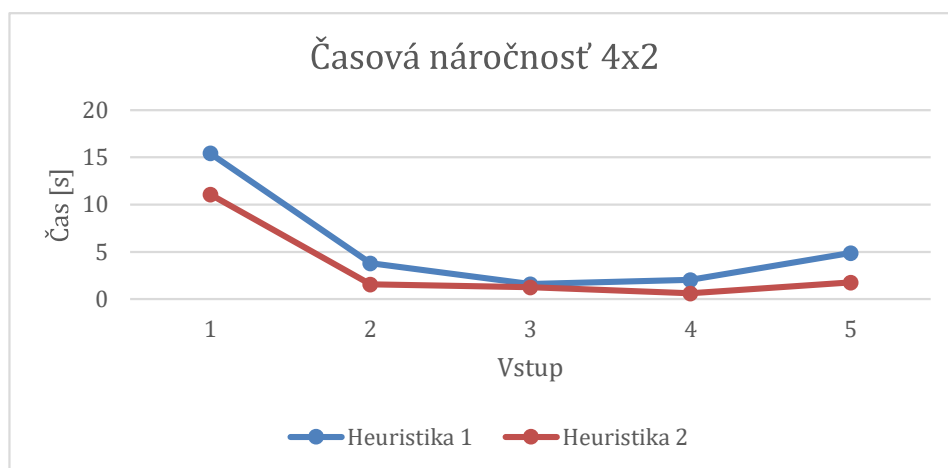
Rozmer 2x3



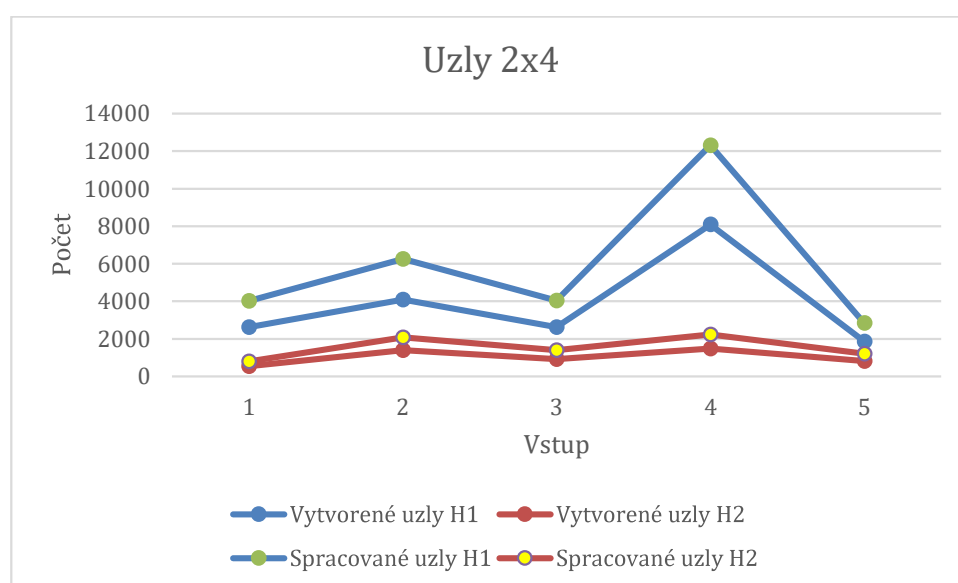
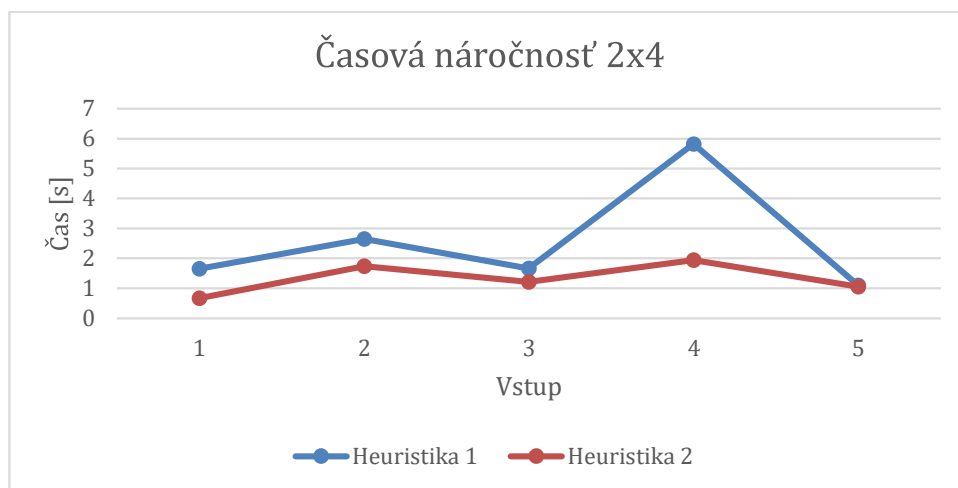
Rozmer 4x2

Aj časová aj pamäťová efektívnosť bola lepšia pri heuristike 2. Čím náročnejšie riešenie, tým väčší je rozdiel medzi použitými heuristikami.

4x2 Uzly				2x4 Uzly					
H1		H2			H1		H2		
Vytvorené	Spracované	Vytvorené	Spracované	Vstup	Vytvorené	Spracované	Vytvorené	Spracované	
17459	26536	7276	10957	1	2634	4027	544	808	
5205	7987	1111	1652	2	4088	6251	1393	2081	
2327	3543	935	1406	3	2623	4034	928	1403	
2884	4429	455	690	4	8093	12309	1491	2240	
6471	9889	1243	1882	5	1867	2840	818	1219	



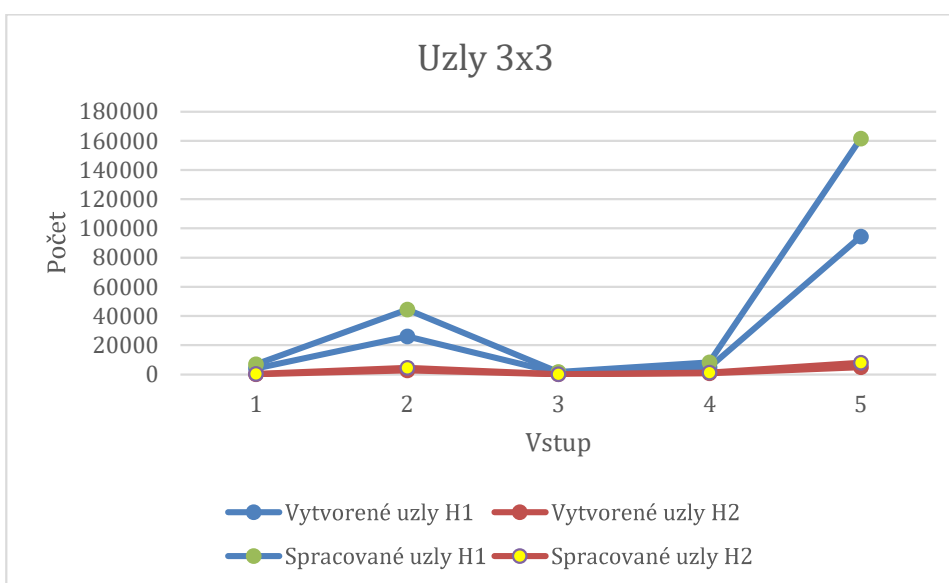
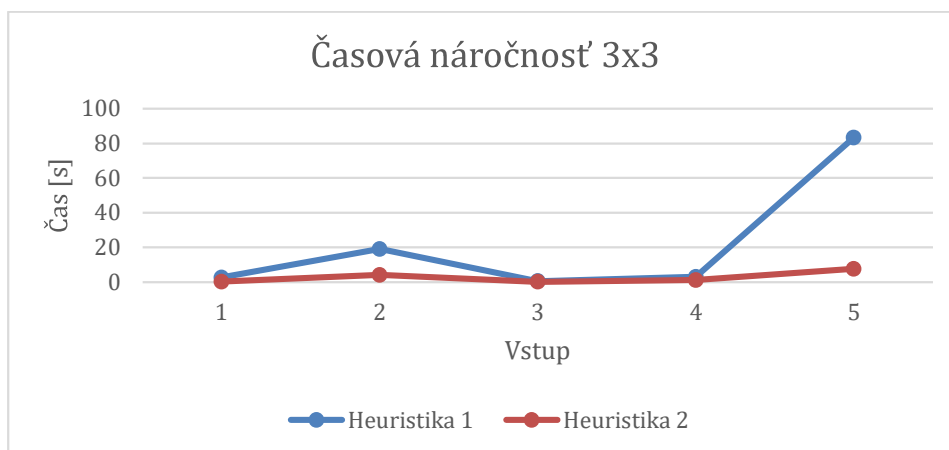
Rozmer 2x4



Rozmer 3x3

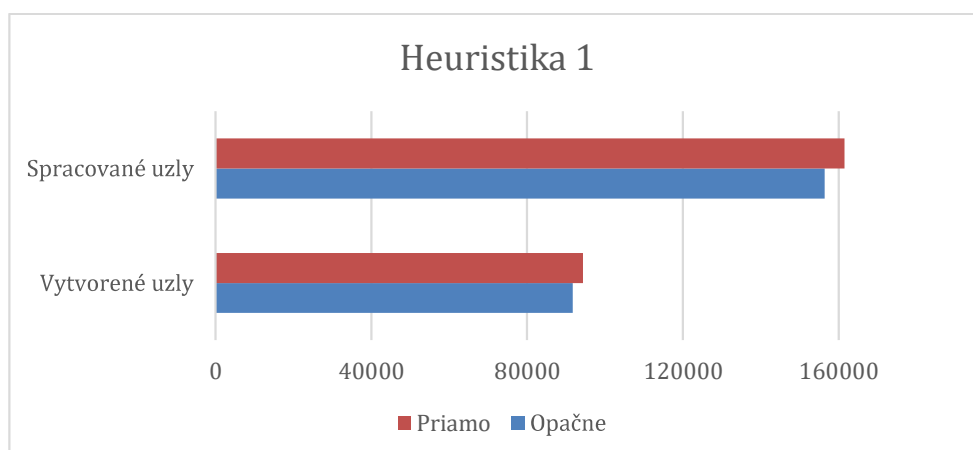
Najzložitejší rozmer, ktorý sme testovali. Časová zložitosť sa pohybovala od niekoľko sekúnd až po niekoľko desiatok sekúnd.

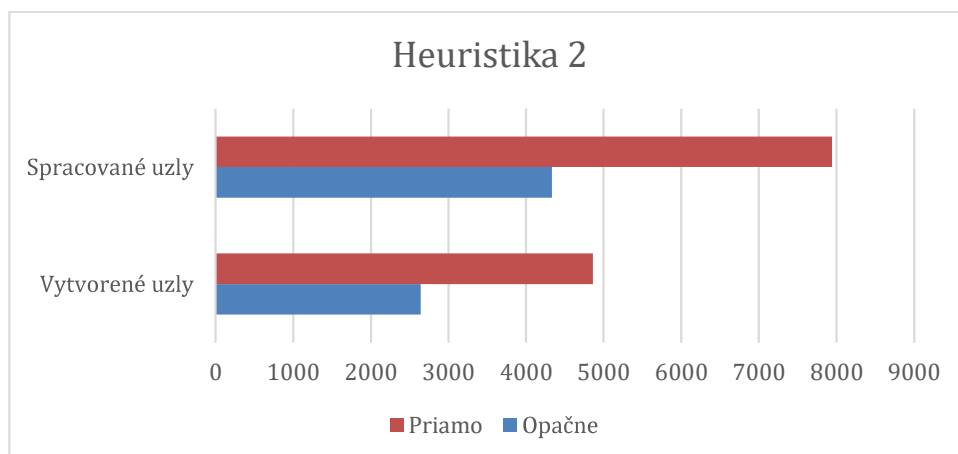
	3x3 Uzly			
	H1		H2	
Vstup	Vytvorené	Spracované	Vytvorené	Spracované
1	4051	7008	139	234
2	25907	44440	2721	4577
3	974	1673	111	190
4	4810	8330	750	1273
5	94332	161502	4862	7943



Výmena začiatku a konca

Túto úlohu sme sa rozhodli riešiť na našom najzložitejšom hlavolame, aby bolo jasne vidieť prípadné rozdiely.







Výmena začiatčného a koncového stavu mala za následok zmenšenie náročnosti. Naš algoritmus našiel skôr cestu z pôvodného konca do pôvodného začiatku, ako priamou cestou. Podobné správanie môžeme očakávať od všeobecného vymieňania začiatku a konca, že jedna cesta, buď priamo alebo opačne, bude menej náročná. Nie nutne to musí byť cesta späť.

Výsledok

Výsledkom programu je postupnosť krokov, spolu s grafickou reprezentáciou stavu po vykonaní predchádzajúceho kroku, aby užívateľ vedel, ako sa stav mení po použitých operáciách.

Výstup vyzerá nasledovne:

HEURISTIC 2 INPUT: 5 3 4 1 2 0 	STEP: 3 Move: Right 0 5 4 1 3 2	STEP: 6 Move: Down 1 0 4 3 5 2	STEP: 9 Move: Right 1 4 2 3 0 5
STEP: 1 Move: Right 5 3 4 1 0 2 	STEP: 4 Move: Up 1 5 4 0 3 2	STEP: 7 Move: Left 1 4 0 3 5 2	STEP: 10 Move: Down 1 0 2 3 4 5
STEP: 2 Move: Down 5 0 4 1 3 2	STEP: 5 Move: Left 1 5 4 3 0 2	STEP: 8 Move: Up 1 4 2 3 5 0	STEP: 11 Move: Right 0 1 2 3 4 5

Time: 0.015

Created nodes: 15

Processed nodes: 23