Slovenská Technická Univerzita

Fakulta informatiky a informačných technológií

Ilkovičova 3, 842 19 Bratislava 4

Richard Szarka

Prehľadávanie stavového priestoru

Zadanie č. 2

Prednášajúci: Ing. Lukáš Kohútka, PhD.

Cvičiaci: Ing. Boris Slíž

Cvičenie: Streda 15:00

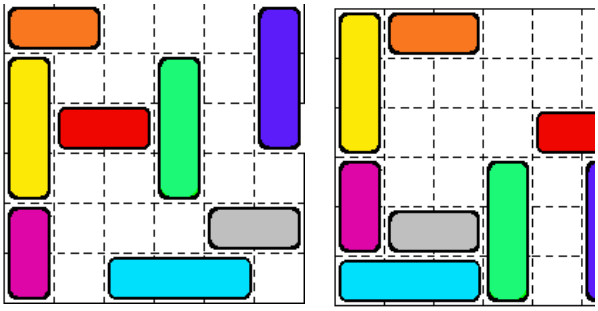
Obsah

1 Definovanie problému

Úlohou je nájsť riešenie hlavolamu **Bláznivá križovatka**. Hlavolam je reprezentovaný mriežkou, ktorá má rozmery 6 krát 6 políčok a obsahuje niekoľko vozidiel (áut a nákladiakov) rozložených na mriežke tak, aby sa neprekrývali. Všetky vozidlá majú šírku 1 políčko, autá sú dlhé 2 a nákladiaky sú dlhé 3 políčka. V prípade, že vozidlo nie je blokované iným vozidlom alebo okrajom mriežky, môže sa posúvať dopredu alebo dozadu, nie však do strany, ani sa nemôže otáčať. V jednom kroku sa môže pohybovať len jedno vozidlo. V prípade, že je pred (za) vozidlom voľných *n* políčok, môže sa vozidlo pohnúť o 1 až *n* políčok dopredu (dozadu). Ak sú napríklad pred vozidlom voľné 3 políčka (napr. oranžové vozidlo na počiatočnej pozícii, obr. 1), to sa môže posunúť buď o 1, 2 alebo 3 políčka.

Hlavolam je vyriešený, keď je červené auto (v smere jeho jazdy) na okraji križovatky a môže sa z nej dostať von. Predpokladajte, že červené auto je vždy otočené horizontálne a smeruje doprava. Je potrebné nájsť postupnosť posunov vozidiel (nie pre všetky počiatočné pozície táto postupnosť existuje) tak, aby sa červené auto dostalo von z križovatky alebo vypísať, že úloha nemá riešenie. Príklad možnej počiatočnej a cieľovej pozície je na obr. 1.

Počiatočná pozícia Cieľová pozícia



Obr. 1 Počiatočná a cieľová pozícia hlavolamu Bláznivá križovatka.

Postupnosť posunov vozidiel z počiatočnej do cieľovej pozície z obr.1 je:

VPRAVO(oranzove, 1), HORE(zlte, 1), HORE(fialove, 1), VLAVO(sive, 3), VLAVO(svetlomodre, 2), DOLE(tmavomodre, 3), DOLE(zelene, 2), VPRAVO(cervene, 3)

Riešenie b)

2 Opísanie zadávania vstupu a zadávania počiatočného stavu

Mapa na ktorej sa hýbu autá je konštantne 6x6. Maximálny počet áut v mojej implementácii je 13. Ak by užívateľ chcel zvýšiť počet áut, musel pri pridať novú farbu do súbora dictionaries.py. Taktiež pridať novú farbu v súbore GUI. Farba by mala mať unikátne počiatočné písmeno pre správne označenie a fungovanie algoritmu.

Príklad vstupného súboru (carsState4.txt):

```
2,1,2,h
3,3,1,v
3,5,0,v
3,2,5,h
2,0,4,v
3,0,1,v
2,0,0,h
2,4,4,h
```

Prvé číslo určuje veľkosť auta. Druhé číslo určuje súradnicu X a tretie určuje súradnicu Y. Písmeno h alebo v určuje či je auto horizontálne alebo vertikálne. Prvé auto v zadanom súbore je vždy červené (auto ktoré chceme dostať von)

Mapa vyzerá nasledovne:

```
  0 1 2 3 4 5
# # # # # # # #
| O O . . . B | 0
| Y . . G . B | 1
| Y R R G . B = 2
| Y . . G . . | 3
| P . . . S S | 4
| P . C C C . | 5
# # # # # # # #
```

Políčka s písmenom značia 1 auto. Políčka s bodkou značia prázdne miesto. Rovná sa označuje východ ktorý je vždy na pozícii [2, 5]. Na obrázku je taktiež vidno súradnicový systém mojej implementácie.

3 Opis výstupu

```
  0 1 2 3 4 5
# # # # # # # #
| O O . . . B | 0
| Y . . G . B | 1
| Y R R G . B = 2
| Y . . G . . | 3
| P . . . S S | 4
| P . C C C . | 5
# # # # # # # #
```

Vykreslený počiatočný stav

Informácie o stackoch, hĺbke a čase

```
47 unprocessed states  | 1074 processed states  | 18 depth Done in 4.958552122116089
```

```
  0 1 2 3 4 5
# # # # # # # #
| Y O O . . . | 0
| Y . . . . . | 1
| Y . . . R R = 2
| P . . G . B | 3
| P S S G . B | 4
| C C C G . B | 5
# # # # # # # #
```

Cieľový stav

```
Orange moves forward
Yellow moves backward
Pink moves backward
Silver moves backward
Silver moves backward
Cyan moves backward
Silver moves backward
Cyan moves backward
Green moves forward
Green moves forward
Red moves forward
Red moves forward
Blue moves forward
Blue moves forward
Blue moves forward
Red moves forward
```

Kroky pomocou ktorých sa dostaneme z počiatočného stavu ku cieľovému

4 Opis jednotlivých tried

Triedy nájdeme v súbore classes.py. Program využíva 2 triedy a to triedu `class Map` a `class Car`.

Atribúty triedy mapy sú nasledovné:

```
class Map:

   def __init__(self, cars, exitPosition, beforeMap, lastMove, depth,
file):
       self.cars = cars # pole áut
       self.exitPosition = exitPosition # pozícia na výstup
       self.beforeMap = beforeMap # rodič
       self.lastMove = lastMove # aký bol krok z rodičia na vytvorenie
tohoto stavu
       self.depth = depth # hĺbka stavu
       self.map = [] # pole vykreslenej mapy
```

Pri inicializovaní prvého stavu sa trieda vytvorí nasledovne:

```
map = Map("X", [5, 2], "root", "root", 0, file)     # inicializovanie prého
stavu
```

Znak “X” značí že autá ešte neboli načítané a preto ich treba načítať zo súboru file. Taktiež neexistuje rodič tohoto stavu preto hodnoty beforeMap a lastMove sú “root”.

Pri ďalších volaniach sa mapy vytvárajú nasledovne:

```
newMap = Map(deepcopy(helpMap.cars), [5, 2], currMap, helpMap.cars[i].color
+ " moves forward", currMap.depth + 1, None)
```

Nová mapa sa vytvorí podľa “hlbokej” kópie poľa áut rodiča. Hlboké kopírovanie zaručí že nové pole áut bude aj iný objekt. Ďalej sa posiela aj rodič a posledný krok (farba auta a typ pohybu). Taktiež hĺbka nového stavu je hĺbka rodiča zväčšená o jedna

Trieda mapy obsahuje ešte funkciu na aktualizovanie vykreslenej mapy, a jej rozoslanie všetkým autám v poli áut (cars). Ak auto spravilo nejaký pohyb, tak treba aktualizovať mapu a poslať nový stav mapy všetkým autám.

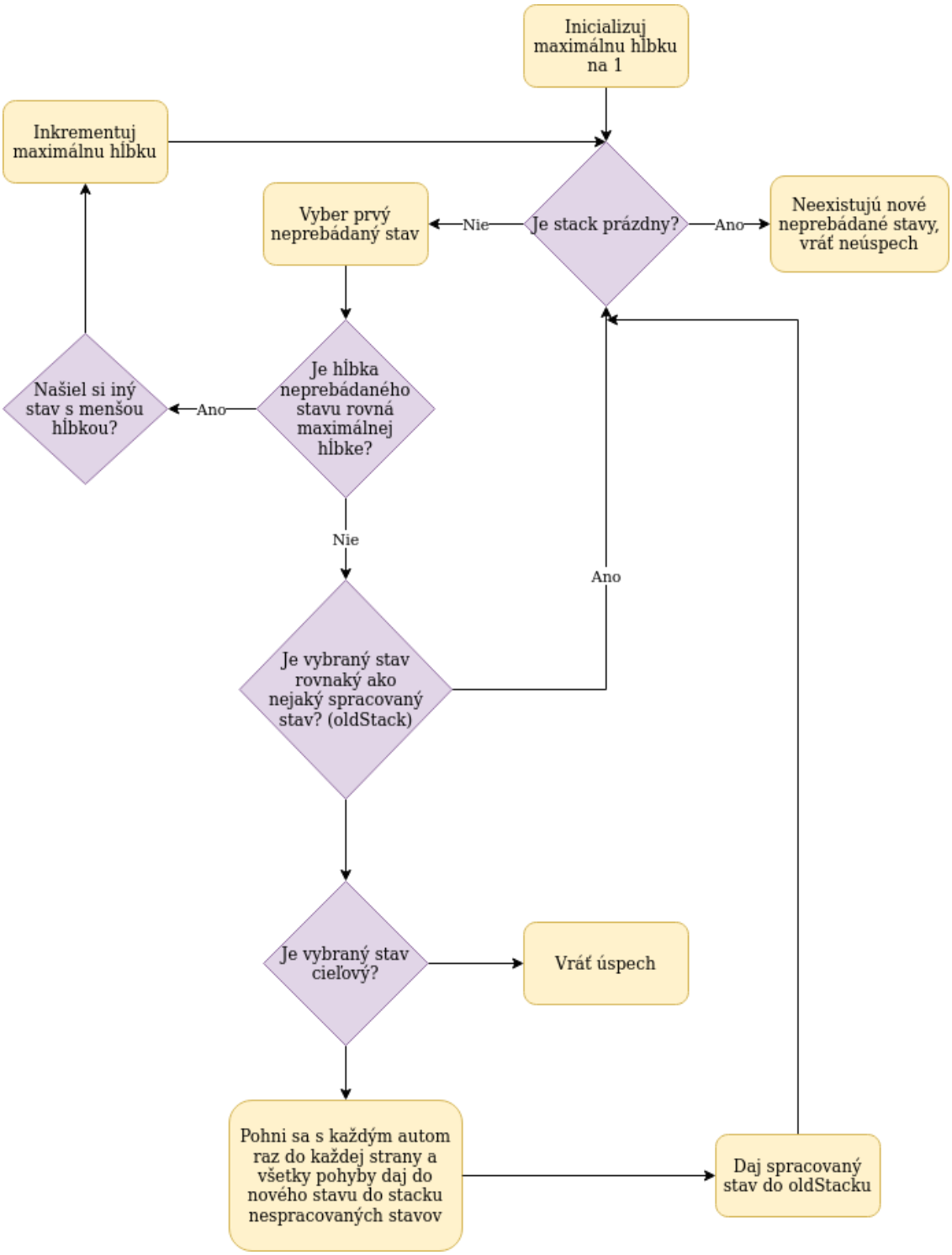
```
class Car:

    # inicializácia auta
    def __init__(self, color, size, position, rotation):
        self.color = color # farba daného auta
        self.size = size # veľkosť auta
        self.postion = position # pozícia auta
        self.rotation = rotation # rotácia auta
        self.map = [] # mapa áut
```

Trieda auta obsahuje všetky potrebné veci na zadefinovanie stavu konkrétneho auta. Atribút mapa je vykreslená mapa v dvojdimenzionálnom poli. Podľa dané auto vie, kde sa nachádzajú ostatné autá. Trieda auta obsahuje ešte dve funkcie, ktoré hýbu s autom dopredu alebo dozadu. Kolízie sa ošetrujú podľa dostupnej mapy áut. Ak sa auto vedelo pohnúť, funkcia vráti hodnotu 1, ináč vráti 0.

Pojem dopredu a dozadu je závislý aj od rotácie auta.

5 Algoritmus cyklicky prehlbujúceho sa hladania a jeho implementácia

Algoritmus si drží nespracované stavy v stacku a mapy spracovaných stavov v oldStacku (“starý stack”). Algoritmus cyklicky prehlbujúce sa hľadania v mojej implementácii funguje nasledovne:



Cyklicky prehlbujúce sa prehľadávanie je pamäťovo šetrnejšie ako prehľadávanie do šírky (je porovnateľné s prehľadávaním do hĺbky), ale na druhú stranu nájde najbližší uzol ku počiatočnému stavu (prehľadávanie do šírky). Cyklicky prehlbujúce sa prehľadávanie je “zlatá stredná cesta” medzi prehľadávaním do širky a do hĺbky, avšak časovo je o trochu pomalšie. Funkcia začína prehľadávať do hĺbky 2 a vždy sa prehlbuje o 2. Funkcia realizujúca prehľadávanie je:
`def iterative_depthSearch(stack, oldstack).`

6 Časové a pamäťové testy

1. **Test - stav zo zadania (carsState4.txt)**

```
  0 1 2 3 4 5
# # # # # # # #
| O O . . . B | 0
| Y . . G . B | 1
| Y R R G . B = 2
| Y . . G . . | 3
| P . . . S S | 4
| P . C C C . | 5
# # # # # # # #


47 unprocessed states  | 1074 processed states  | 18 depth Done in 4.591583967208862



  0 1 2 3 4 5
# # # # # # # #
| Y O O . . . | 0
| Y . . . . . | 1
| Y . . . R R = 2
| P . . G . B | 3
| P S S G . B | 4
| C C C G . B | 5
# # # # # # # #
```

Stav zo zadania našlo pre prebádaní 1074 unikátnych stavov. Na zásobníku ostalo 47 nespracovaných stavov. Nájdený stav bol v Hĺbke 16 alebo 17. Zabraná pamäť bola 15MB.

Riešenie za zobrazí aj prostredníctvom GUI a program vám ukáže riešenie a zároveň sa vypíšu potrebné kroky do terminálu

2. **Test - vymyslený stav (carsState1.txt)**

```
  0 1 2 3 4 5
# # # # # # # #
| . . B . M M | 0
| . . B . P P | 1
| R R B . O G = 2
| . . . . O G | 3
| S Y Y Y . G | 4
| S . . C C C | 5
# # # # # # # #


2794 unprocessed states  | 5441 processed states  | 26 depth Done in 66.80757641792297



  0 1 2 3 4 5
# # # # # # # # |
| . . M M O . | 0
| P P B . O . | 1
| . . B . R R = 2
| . . B . . G | 3
| S . Y Y Y G | 4
| S . C C C G | 5
# # # # # # # #
```

Vidíme, že pri tomto riešení sa musel vnoriť do väčšej hĺbky a to je 26. Prebádané unikátne stavy boli 5441 a alokovaná pamäť bola 33MB.

3. **Test - vymyslený stav(carsState8.txt)**

```
  0 1 2 3 4 5
# # # # # # # #
| G G Y Y Y N | 0
| B B O . . N | 1
| C . O R R N = 2
| C . . M K K | 3
| P . . M . . | 4
| P S S L L L | 5
# # # # # # # #
```

```
205 unprocessed states  | 781 processed states  | 52 depth Done in 4.743866205215454
```

```
  0 1 2 3 4 5
# # # # # # # #
| C G G Y Y Y | 0
| C . O . B B | 1
| P . O . R R = 2
| P K K M . N | 3
| . . . M . N | 4
| S S L L L N | 5
# # # # # # # #
```

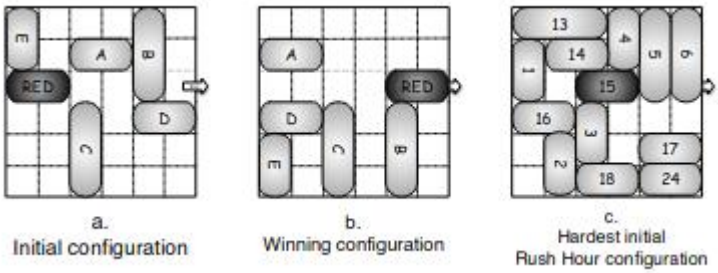
V teste č. 3 vidíme, že algoritmus preskúmal až po 52. hĺbku pričom bolo len 781 unikátnych stavov. Môžeme konštatovať že sa tak stalo z dôvodu nízkeho počtu možností na pohyb jednotlivých áut. Riešenie našiel za 4.73s. Využil 15MB pamäte.

4. **Test - vymyslený stav (carsState7.txt)**

```
  0 1 2 3 4 5
# # # # # # # #
| G P Y Y K . | 0
| G P O . K F | 1
| G . O R R F = 2
| B B B M . F | 3
| . . S M L L | 4
| C C S N N . | 5
# # # # # # # #


1326 unprocessed states  | 3062 processed states  | 84 depth Done in 29.746827602386475



  0 1 2 3 4 5
# # # # # # # #
| Y Y O M K . | 0
| . . O M K . | 1
| G . . . R R = 2
| G P B B B F | 3
| G P S L L F | 4
| C C S N N F | 5
# # # # # # # #
```

V teste číslo 3 je vidno, že hĺbka je až 84. Na druhú stranu vidíme že unikátnych stavov je len 3062, čo znamená, že v tomto prípade boli na rovnakých hĺbkach opakujúce sa stavy. Riešenie našiel za 29.74 sekundy. Zabratá pamäť sa rovnala 21,9 MB.

5. **Test - najťažší možný stav(carsState9.txt)**



Zdroj: http://di.ulb.ac.be/algo/secollet/papers/crs06.pdf

```
  0 1 2 3 4 5
# # # # # # # #
| G G G M N K | 0
| B Y Y M N K | 1
| B . R R N K = 2
| C C O . . . | 3
| . P O . L L | 4
| . P S S F F | 5
# # # # # # # #
```

```
7692 unprocessed states  | 19300 processed states  | 94 depth Done in 779.4208273887634
```

```
  0 1 2 3 4 5
# # # # # # # #
| B P G G G . | 0
| B P Y Y . . | 1
| . . . . R R = 2
| C C O M N K | 3
| L L O M N K | 4
| S S F F N K | 5
# # # # # # # #
```

V tomto prípade musel prehľadať až skoro 20000 unikátnych stavov a alokoval 92MB. Riešenie našiel najlepšie v hĺbke 94. Riešenie našiel za 779 sekúnd

Vidíme, že rýchlosť nájdenia cieľového stavu závisí od počtu prebádaných stavov. Nielen, že sa počet stavov vyššou hĺbkou zväčšuje, ale aj počet porovnaní.

6. **Test - stav bez riešenia (carsState2.txt)**

```
  0 1 2 3 4 5
# # # # # # # #
| . . . P Y O | 0
| . . . P Y O | 1
| R R . G G G = 2
| . B C . . . | 3
| . B C . . . | 4
| . . . . . . | 5
# # # # # # # #


0 unprocessed states  | 78 processed states  | 14 depth  no solution
Done in 0.19721508026123047
```

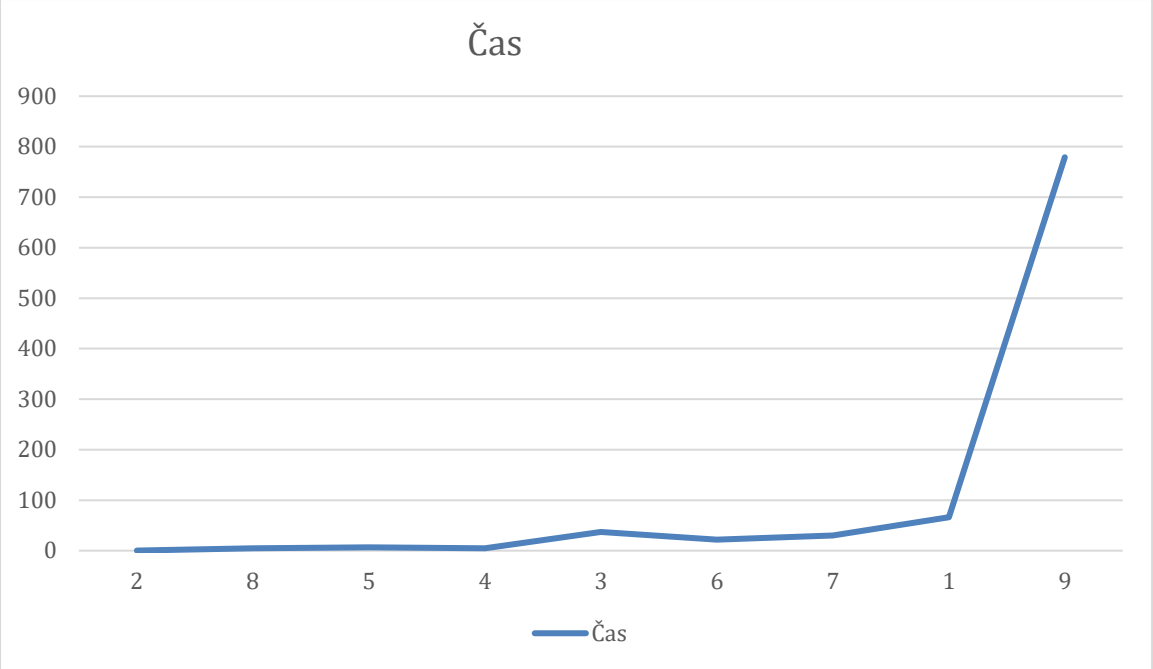
V tomto prípade vidíme, že červené auto (RR) blokuje zelené auto (GGG). Tým pádom sa červené auto nikdy nemôže dostať von. Zistenie, že dané riešenie neexistuje nastane vtedy, ak už neexistuje neprebádaný unikátny stav. Tým pádom vieme povedať že existuje 78 stavov (konfigurácii) hracej plochy s takto umiestnenými autami a žiadny z nich nie je cieľový stav.

Program zabral 8MB a zistenie neexistencii cieľového stavu zistil za 0.19s.

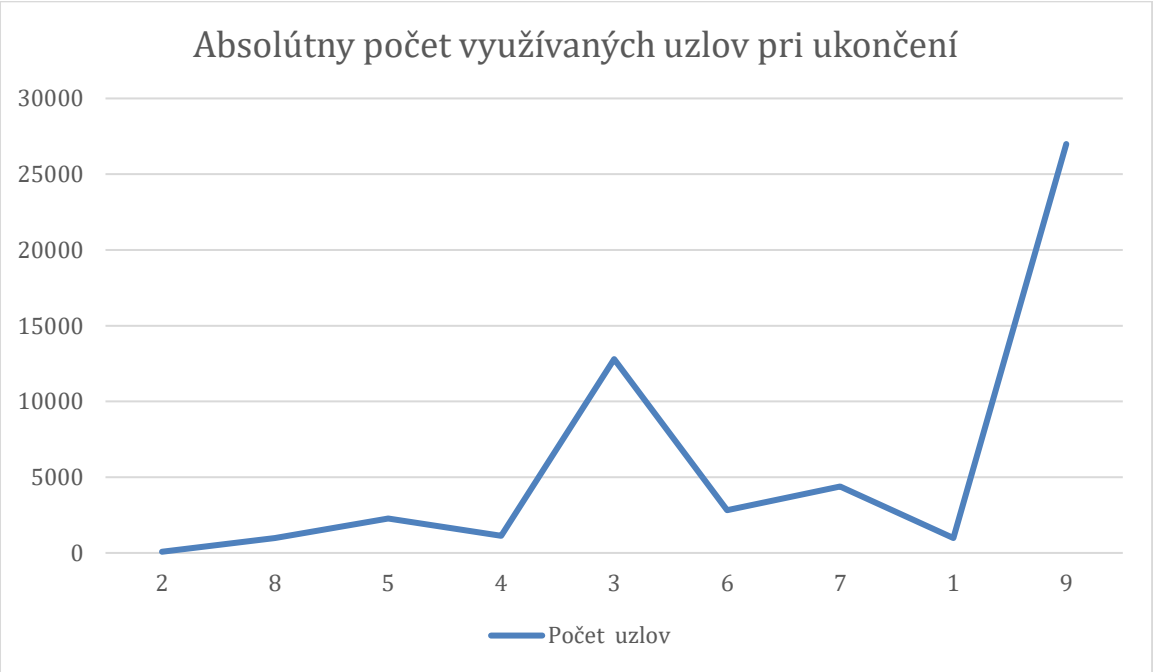
7 Zhodnotenie testov

	Stav stacku s nespracovanými uzlami pri ukončení	Stav stacku spracovaných uzlov pri ukončení	Čas [s]	Pamäť [MB]
carsState2	0	78	0,19	8
carsState8	205	781	4,74	15
carsState5	1313	962	6,88	16,4
carsState4	47	1074	4,55	15
carsState3	10468	2330	36,86	58,6
carsState6	323	2498	21,48	26,1
carsState7	1326	3062	29,74	21,9
carsState1	2794	5441	0,19	33
carsState9	7692	19300	779	92

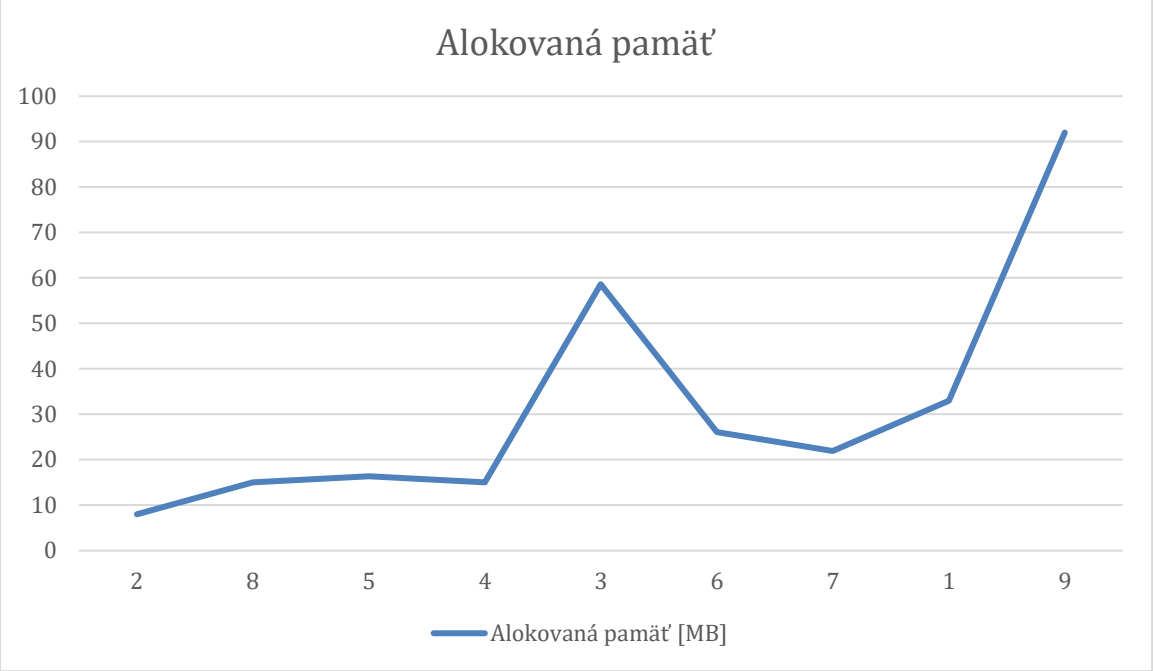
Vidíme, že zložitosť riešení od ľahkého po ťažšie narastá exponenciálne. Je to aj tým, že čím je riešenie zložitejšie, tým hlbšie musí ísť algoritmus prehľadávať. Na hĺbke nezáleží z toho dôvodu, že ak sa v hĺbkach konfigurácie stavov opakujú, tak sa jednotlivé hĺbky spracúvajú rýchlejšie. Avšak s narastajúcou hĺbkou pribúda aj počet uzlov na danej vrstve.



Vidíme, že graf s časovými meraniami dostatočne kopíruje graf s unikátnymi prebádanými uzlami. Rozdiel môžeme vidieť len v prípadoch 7 a 1. Rozdiel je v tom že, avšak unikátnych stavov bolo v prípade 7 a 1 viac ako v prípade 3, čas na nájdenie cieľového stavu bol menší alebo primerane rovnaký. Rozdiel nastal z dôvodu, že v prípadoch 7 a 1 je viacej unikátnych stavov ale algoritmus nemusel spracovávať tak veľa rovnakých stavov.



V grafe je znázornený absolútny počet uzlov.



Alokovaná pamäť dostatočne kopíruje graf s absolútnymi počtami uzlov. Jediný viditeľný rozdiel je v prípade 1. Dôvod kvôli akému mohlo nastať vychýlenie, je že dané riešenie našiel na konci prehľadávanej vrstvy (o chvíľu by sa prehlboval do väčšej hĺbky) a trash collector ešte neuvoľnil nevyužívané uzly.

8 Záver

V tomto zadaní sme mali vytvoriť algoritmus prehľadávania stavov. Moja implementácia obsahuje iteratívnu cyklickú formu prehľadávania do hĺbky. Algoritmus prehľadáva do hĺbky ale neprechádza ďalej po určitú úroveň (hĺbku). Ak prebádal všetky stavy po danú hĺbku tak sa prehĺbi. Algoritmus je kompromis medzi algoritmom prehľadávania do hĺbky a prehľadávania do šírky. Pamäťovo je zhovievavejší ako prehľadávanie do šírky a zároveň si udržiavame vlastnosť, že sa nájde riešenie v čo najnižšej hĺbke. Avšak algoritmus je jemno časovo menej efektívny kvôli prehlbovaniu.

Moje riešenie nájde vždy riešenie ak existuje a ak neexistuje zistí, že riešenie neexistuje. Triedy majú len také parametre, ktoré sú nevyhnutné. Algoritmus našiel aj najťažšie možné riešenie (carsState9.txt) čím sme si overili, že každé riešenie nájde v rozumnom čase (najťažšie bolo vyše 10min, priemer bol okolo 1min, klasické riešenia do niekoľko sekúnd).

Algoritmus som implementoval v jazyku python čo zapríčiňuje pomalšie a pamäťovo menej výhodné výsledky. Avšak na jazyk python sú aj tieto výsledky dobré. Zlepšenie v jazyku python by bolo výmenou tried za dataclass (implementované z jazyka C), čím by si python nemusel vytvárať informácie o triedach a prístup k nim by bol rýchlejší. Avšak použitie tried uľahčuje prípadnú budúcu doimplementáciu.

Časové zložitosti:
V – faktor vetenia sa
D – hĺbka

Časová zložitosť $O(D^V)$
Pamäťová zložitosť $O(DV)$