Slovenská technická univerzita

Fakulta informatiky a informačných technológii

Ilkovičova 3, 812 19 Bratislava

**Umelá inteligencia**

**2.** **Zadanie – Prehľadávanie stavového priestoru**

**Michal Ostrodický**

Cvičiaci: Ing. Ivan Kapustík

Študijný odbor: Informatika

Ročník: 2. Bc

Akademický rok: 2016/2017

Obsah

[Znenie zadania: 3](#_Toc477694254)

[Opis riešenia: 4](#_Toc477694255)

[Reprezentácia údajov: 4](#_Toc477694256)

[ Trieda Vozidlo 4](#_Toc477694257)

[ Trieda Uzol 4](#_Toc477694258)

[Algoritmus: 5](#_Toc477694259)

[ Trieda Operator 5](#_Toc477694260)

[ Trieda BFS 5](#_Toc477694261)

[ Trieda DFS 6](#_Toc477694262)

[ Trieda prehladavanie 6](#_Toc477694263)

[Testovanie 6](#_Toc477694264)

[ Vstup1 7](#_Toc477694265)

[ Vstup2 7](#_Toc477694266)

[ Vstup3 8](#_Toc477694267)

[ Vstup4 8](#_Toc477694268)

[ Vstup5 8](#_Toc477694269)

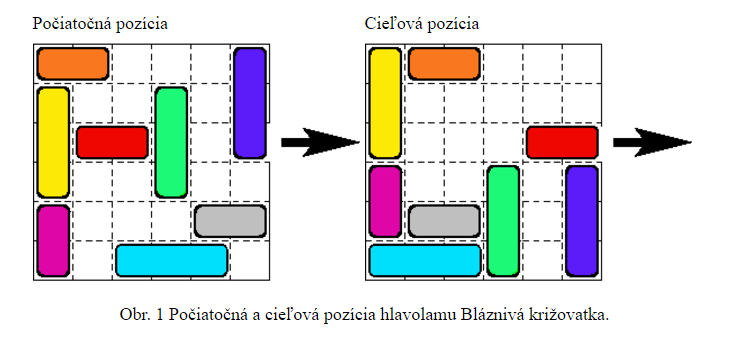
[ Vstup6 8](#_Toc477694270)

[Zhodnotenie: 9](#_Toc477694271)

[**Porovnanie vlastností**: 9](#_Toc477694272)

# Znenie zadania:

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 z nej teda 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.

**Problém 1.** Použite algoritmus prehľadávania do šírky a do hĺbky. Porovnajte ich výsledky.

# Opis riešenia:

Zadanie som riešil algoritmami neinformovaného hľadania – prehľadávaním do šírky(Breadth first search - BFS) a do hĺbky(Depth first search - DFS). Prehľadávanie do šírky používa dátovú štruktúru radu(Queue) a princíp FIFO, čiže First in First out. Ten stav, ktorý bol vložený do frontu najskôr bude pri nasledujúcom výbere z frontu vybraný ako prvý. Nové uzly sa pridávajú na koniec frontu.

Dátovú štruktúru zásobník(Stack) som použil v prehľadávaní do hĺbky. Stack je založený princípe LIFO, last in first out. Ten stav, ktorý sme pridali do zásobníka ako posledný bude pri nasledujúcom výbere vybraný ako prvý. Nové uzly sa pridávajú na začiatok zásobníka.

Obe prehľadávania generujú veľa stavov a mnohokrát sa tieto stavy opakujú. V mojom riešení som použil Hashmapu na uchovávanie stavov, ktoré som už vytvoril, aby som vedel jednoznačne povedať, či som daný stav už niekde vyššie v strome prehľadávania nevygeneroval. Ako som už spomínal vyššie, v BFS som používal štruktúru rad na uloženie ešte nespracovaných stavov, a v DFS štruktúru zásobníka.

Dôležitou časťou v riešení je použitie vhodných operátorov na nájdenie cieľového stavu. Moje riešenie používa 4 operátory a to pohyb vozidla – vpravo, vľavo, dole a hore.

Cieľová pozícia je dosiahnutá vtedy ak sa červené vozidlo(vždy prvé v poradí) dostane na x-ovú súradnicu 5 v hociktorom riadku križovatky.

Činnosť algoritmov prehľadávania sa vo všeobecnosti veľmi podobajú, oba spôsoby vyhľadávania majú rôzne výhody a nevýhody. V nasledujúcich kapitolách sa pokúsim porovnať tieto dve hľadania.

# Reprezentácia údajov:

V tejto kapitole popíšem triedy, ktoré používam na reprezentáciu údajov. Dôležité sú ich atribúty, ktorými reprezentujem stav.

## Trieda Vozidlo

Predstavuje štruktúru v ktorá reprezentuje informácie o jednom vozidle v križovatke.   
Má atribúty:

* + Farba – String v ktorom je uložená farba vozidla
  + Velkost – predstavuje veľkosť vozidla. Osobné majú veľkosť 2, nákladné majú veľkosť 3.
  + SuradnicaY – Integer na uloženie y-ovej súradnice
  + SuradnicaX – Integer na uloženie x-ovej súradnice
  + Posun – Znak, ktorý môže byť h alebo v. Znamená nasmerovanie vozidla v križovatke. Ak je vozidlo horizontálne(h), tak sa môže posunúť len doprava a doľava. Ak je vertikálne (v) tak je možný posun hore alebo dole.

Moja reprezentácia mapy je urobená tak, že **súradnica x určuje stĺpec a y určuje riadok**.

## Trieda Uzol

Štruktúra v ktorej je uložené aktuálne rozloženie vozidiel stavu. Dôležitým atribútom je teda zoznam vozidiel(poleVozidiel), pre každé jedno vozidlo mám uložené jeho atribúty.

Ďalšie atribúty triedy:

* + hashStavu – číslo, ktoré je vygenerované metódou getHashCode. Je to hash kód daného rozloženia vozidiel v stave. Tento hash nie je jednoznačným určením stavu, keďže viacero stavov môže mať rovnaký hash kód.
  + hashPredchodcu – Hash predchádzajúceho stavu, od ktorého som sa dostal do stavu, s ktorým momentálne pracujem. Je to číslo vygenerované metódou getHashCode.
  + IdStavu – je to jednoznačný identifikátor stavu. Pri každom vytvorení nového stavu sa toto číslo zväčšuje a je možné vďaka nemu spätne zreťaziť stavy v poradí v akom boli navštívené.
  + idPredchodcu – ID predchádzajúceho stavu, stavu z ktorého som sa dostal do stavu, s ktorým momentálne pracujem.

getHashCode - Vlastná metóda na vytvorenie hashu pre stav. Do hashu sa započítavajú všetky atribúty poľa vozidiel. Pri zníženie kolízii používam prvočísla 17 a 31 pri výpočte.

vypisVozidiel() – Metóda na výpis stavu, presné rozloženie vozidiel, ich pozície a smer.

# Algoritmus:

Nasledujúce triedy sú dôležité v algoritmoch hľadania do šírky a do hĺbky. Pri samotnom výpočte využívam ich metódy.

## Trieda Operator

Trieda, ktorá obsahuje metódy na vytvorenie stavov pomocou operátorov. Konkrétne operaciaVpravo, operaciaVlavo, operaciaDole, operaciaHore.

Všetky metódy dostavajú rovnaké parametre a to stav, z ktorého boli vytvorené - indexVozidla, ktoré sa pohlo;

posunNaPoziciu - presná x alebo y pozícia na ktorú sa vozidlo dostane posunutím;

posun - počet políčok, ktoré vozidlo prešlo;

idAktualnehoStavu - jedinečné ID, ktoré vytvorený stav dostane.

## Trieda BFS

V tejto triede je algoritmus prehľadávania do šírky. Metóda vypocetBFS začínam tým, že z počiatočného rozloženia vozidiel, ktorý dostane v parametri metódy vytvorí počiatočný stav. Alokuje sa pamäť pre dátové štruktúry Queue(FIFO) a HashMap. Počiatočný stav sa do nich vloží, a algoritmus prejde do slučky while, v ktorej je podmienka, či je front prázdny. Ak front nie je prázdny tak vyberie z neho najspodnejší stav(stav ktorý bol vložený najskôr). Tento stav sa porovná, či nie je cieľový stav.

Ak sa X-ová súradnica červeného auta rovná 5, tak som našiel cieľový stav. Prehľadávaním do šírky som našiel určite najkratšiu cestu. Túto cestu vypíšem tak, že do zoznamu operátorov pridávam operátory, ktoré som použil. V ďalšom while cykle sa cyklím pokým som neprišiel do začiatočného stavu. Zistím si hash a id predchádzajúceho stavu, vyhľadám si ho v hash mape a ak sa rovná ID predchádzajúceho stavu s porovnávaným stavom v hash mape tak si operátor porovnávaného stavu uložím do zoznamu operátorov. Takto sa to pekne reťazí od cieľového až do začiatku(ktorý má ID rovný 0). Každý stav teda má svoje ID a ID svojho predchodcu. Pri prechode medzi stavmi si do zoznamu operátorov pridávam atribút poslednePouzityOperator, ktorý má tvar napr. VPRAVO(Oranzove, 1). Ak som prišiel do začiatočného stavu, postupnosť operátorov vypíšem.

Ak som nenašiel cieľovú pozíciu, pre každé vozidlo v poli vozidiel skúšam použiť operátor. Použitím operátora(v poradí: vpravo, vľavo, dole, hore) sa vytvorí nový stav s už novým rozložením vozidiel. To či môžem vykonať operátor závisí od dvoch podmienok, ktoré musia byť splnené.

1. Vozidlo sa nemôže posunúť ak mu bráni nejaké iné vozidlo v pohybe.
2. Vozidlo nemôže vyjsť z križovatky.

Prvú podmienku viem sledovať tým, že si vytvorím pole obsadenosti mapy z núl a jednotiek – metóda vytvorPole. Ak po použití operátor sa chcem dostať na políčko v mape, kde je 0, všetko je v poriadku a takýto operátor môžem použiť. Ak sa dostanem na 1 v mape, tak nie je splnená prvá podmienka a tým sa celé použitie operátora na dané vozidlo zruší.

Druhú podmienku kontrolujem tak, že nová súradnica vozidla po presune môže byť len 1 až 5 pre vozidlá s veľkosťou 2 a 1 až 4 pre vozidlá s veľkosťou 3.

Vytvorenie nového stavu operátorom zabezpečujú statické funkcie v triede Operator. Vytvorený stav sa pokúsim vyhľadať v hash mape, ak je už vytvorený hash kód pre takýto stav musím porovnať či je naozaj totožný s existujúcim. Môže sa stať, že 2 stavy s rôznym rozložením vozidiel majú rovnaký hash kód. Toto overujem v pomocnej metóde zistiTotoznost , ktorá ako parameter dostáva 2 stavy a vracia true ak sú rovnaké inak false. Ak sa teda vytvoril stav, ktorý nie je v hash mape, pridá sa do nej a rovnako aj do radu nespracovaných uzlov.

V prípade, že sa nedá použiť operátor s vozidlom sa nič neurobí.

Takto sa prejde celý zoznam vozidiel a vygenerujú sa nové stavy. Algoritmus sa vráti naspäť k podmienke či je front neprázdny.

Ak je prázdny, riešenie v bláznivej križovatky **neexistuje**. Metóda vypocetBFS vráti cieľový stav ak sme našli riešenie alebo null ak riešenie neexistuje.

## Trieda DFS

V tejto triede je algoritmus prehľadávania do hĺbky. Metóda vypocetDFS začínam tým, že z počiatočného rozloženia vozidiel, ktorý dostane v parametri metódy vytvorí počiatočný stav.

Na rozdiel od BFS sa tu použije dátová štruktúra Stack(LIFO), ktorá pri výbere zo zásobníka vyberá stav, ktorý bol pridaný ako posledný. Výpis cesty pomocou predchádzajúcich ID je pri DFS nepoužiteľné a preto algoritmus vracia len cieľový stav, ktorý našiel.

Všetky ostatné časti algoritmu sú totožné s prehľadávaním do šírky(BFS).

Je teda zachovaná akási všeobecná štruktúra prehľadávacích algoritmov:

1. Vytvor počiatočný uzol a umiestni medzi vytvorené a zatiaľ nespracované uzly
2. Ak neexistuje žiadny vytvorený a zatiaľ nespracovaný uzol, skonči s neúspechom - riešenie neexistuje
3. Vyber najvhodnejší uzol z vytvorených a zatiaľ nespracovaných, označ ho aktuálny
4. Ak tento uzol predstavuje cieľový stav, skonči s úspechom - vypíš riešenie
5. Vytvor nasledovníkov aktuálneho uzla a zaraď ho medzi spracované uzly
6. Vytrieď nasledovníkov a ulož ich medzi vytvorené a zatiaľ nespracované
7. Choď na krok 2.

## Trieda prehladavanie

V tejto triede si používateľ na začiatku zadá názov testovacieho vstupu, ktorý chce spustiť. Zobrazí sa mu križovatka pred posunom. Čísla vozidiel zodpovedajú poradovému číslu ako boli zapísané vo vstupnom súbore.

Zavolá sa metóda vypocetBFS a vypocetDFS, pre každú z nich zobrazím počet prejdených stavov, mapu cieľového stavu a dĺžku času.

# Testovanie

Počas programovania algoritmov som používal viacero testovacích vstup, ktoré som uložil do textových súborov, aby som ich nemusel stále ručne vkladať alebo kopírovať do programu.

Prehľad testovacích stavov, ktoré prikladám aj k zdrojovému kódu.

## Vstup1

Na vstupe sú 3 vozidlá (pre upresnenie – prvá je farba, druhá je veľkosť, tretia je Y-ová súradnica(riadok), štvrtá je X-ová súradnica(stĺpec) a posledný je smer).

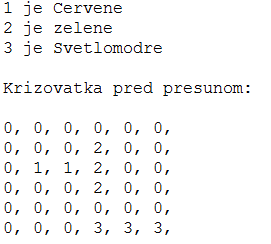
Cervene 2 3 2 h

zelene 3 2 4 v

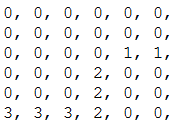
Svetlomodre 3 6 4 h

Na tomto vstupe som si overoval správnosť operátorov, či sa správne vozidlá posúvajú. Tento vstup patrí medzi jednoduchšie s postupnosťou 3 operátorov(VLAVO(Svetlomodre, 3) DOLE(zelene, 2) VPRAVO(Cervene, 3). Počet prešetrených stavov pri BFS je 29 a pri DFS 10. Obe prehľadávania nájdu totožný stav križovatky. BFS čas výpočtu je 6 ms.

DFS čas výpočtu 1 ms.



Po presune:



## Vstup2

Cervene 2 3 2 h

Oranzove 2 1 1 h

Zlte 3 2 1 v

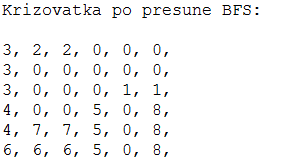
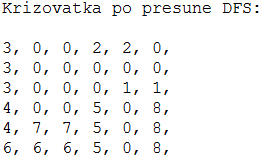
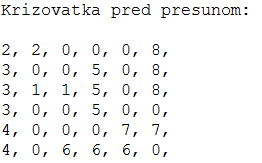
Fialove 2 5 1 v

Zelene 3 2 4 v

Svetlomodre 3 6 3 h

Sive 2 5 5 h

Tmavomodre 3 1 6 v

Toto je vstup zo zadania. Tento je špecifický v tom, že generuje veľmi veľa rovnakých stavov. Na tomto vstupe som overoval funkčnosť hashovacej mapy. Počet operátorov je 8 a ich použitie je rovnaké ako je na stránke v ukážkovom príklade. V mojom programe sú niektoré vymenené v poradí, no celkový výsledok to nemení. Je to zapríčinené tým, ako skúšam operátory. Používajú sa v poradí doprava, doľava, dole a hore. Postupnosť operátorov: VPRAVO(Oranzove, 1) HORE(Zlte, 1) HORE(Fialove, 1) VLAVO(Svetlomodre, 2) VLAVO(Sive, 3) DOLE(Zelene, 2) DOLE(Tmavomodre, 3) VPRAVO(Cervene, 3)

Počet prejdených stavov v prehľadávaní do šírky = 1058

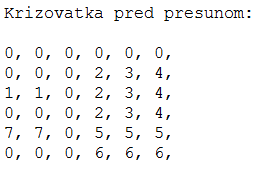
Počet prejdených stavov v prehľadávaní do hĺbky = 252

Priemerný čas výpočtu BFS 48 ms.

Priemerný čas výpočtu DFS 11 ms.

## Vstup3

Tento vstup je špeciálny vstup v tom, že nemá riešenie. Na základe tohto vstupu som vylaďoval program pre neriešiteľné križovatky. Počet prejdených stavov v BFS a DFS je rovnaký =248. Priemerný čas výpočtu BFS je 25 ms a DFS čas výpočtu je 12 ms.



Ďalšie testovacie vstupy spomeniem v krátkosti.

## Vstup4

Ďalší test na negatívny scenár. V tomto vstupe som testoval ako sa správa program, keď je na cieľovej pozícii auto, ktoré sa odtiaľ nemôže inak dostať. Program správne vyhodnotí, že neexistuje riešenie.

## Vstup5

Komplexnejší pozitívny scenár, v križovatke je 10 vozidiel. Počet operátorov potrebných na dosiahnutie cieľovej pozície je až 9. Počet prejdených stavov na nájdenie najkratšej cesty je dvojnásobok ukážkového príkladu – 2138.

## Vstup6

Testovanie či program nájde cieľovú pozíciu aj na inom riadku. Vo vyššie uvedených scenároch bola pozícia červeného auta vždy v riadku 3. Tento vstup má červené auto vo 4. riadku. Cieľová pozícia je nájdená a funguje vtedy ak červené auto dosiahne x-ovú súradnicu 5 v hociktorom riadku v križovatke.

# Zhodnotenie:

Moja implementácia prehľadávania do šírky a do hĺbky je rýchla a efektívna. Medzi ďalšie výhody považujem vykreslenie  nájdeného riešenia. Nájdenie najkratšej cesty prehľadávaním do šírky funguje spoľahlivo.

Ako nevýhodu môžem považovať chýbajúce GUI pre program, kde by som mohol ukázať simuláciu operátorov na vozidlách. Ako ďalšiu uvádzam chýbajúci výpis operátorov pri hľadaní do hĺbky. Riešenie cez ID aktuálneho a predchádzajúceho stavu pri DFS nemá význam, pretože vypisuje strašne dlhú cestu kým príde do cieľového stavu.

Vytvorený jar súbor je spustiteľný na hocijakom počítači, ktorý má nainštalovanú Javu. Pre správne fungovanie program je potrebné, aby boli **vstupné súbory v rovnakom priečinku** ako je súbor.

## **Porovnanie vlastností**:

Časová zložitosť a priestorová zložitosť prehľadávania do šírky je O(b^d), kde b je vetviaci faktor a d je hĺbka najplytšieho cieľového uzla. Faktor vetvenia je maximálny počet nasledovníkov ktoréhokoľvek stavu. Hĺbka najplytšieho cieľového uzla je minimálna dĺžka cesty medzi počiatočným a cieľovým stavom. Hľadanie do šírky je úplne a prípustné.

Časová zložitosť je O(b^m), kde b je vetviaci faktor a m je maximálna hĺbka listového uzla. Na rozdiel od prehľadávania do šírky, priestorová zložitosť prehľadávania do hĺbky je len O(bm) alebo O(m). Hľadanie do hĺbky je úplné iba pre konečný strom hľadania a nie je prípustné.

Hľadanie do šírky je teda úplne a prípustné, ale má vysokú pamäťovú zložitosť. Na druhej strane je prehľadávanie do hĺbky pamäťovo efektívnejšie(nepotrebuje si pamätať všetky uzly v úrovni v ktorej sa nachádza) ale nie je úplne ani prípustné.

V závislosti od dát a čo sa má hľadať vedia byť obe vyhľadávania užitočné. Ak sú stavové stromy veľmi hlboké je efektívnejšie použiť DFS na nájdenie cieľového stavu. Ak sa však hľadá stav, ktorý je bližšie k vrcholu stromu, je lepšie použiť BFS. BFS vie zaručene nájsť najkratšiu cestu v strome na rozdiel od DFS, ktoré nemusí nájsť práve najkratšiu.

V mojom riešení sa potvrdili vlastnosti týchto algoritmov. DFS prechádza rýchlejšie vďaka nižšej pamäťovej zložitosti. V mojich testovacích vstupoch, kde existovalo riešenie prebehlo DFS približne 4x rýchlejšie. Je to však skreslené tým, že pri DFS nehľadám spätne operátorov, čo môže ušetriť pár desiatok ms. Pri vstupoch, kde je stavový strom veľmi malý(do 5 stavov), DFS a BFS trvali identicky dlho.