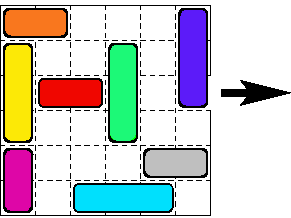
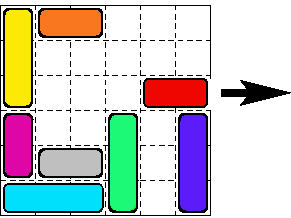
**Zadanie č. 2**

**Prehľadávanie stavového priestoru**

**Zadanie:**

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.

**Opis riešenia:**

Pri riešení som použil programovací jazyk JAVA. Zadanie som mal riešiť princípom cyklicky sa prehlbujúceho vyhľadávania(IDS) . Algoritmus je teda riešený ako prehľadávanie do hĺbky až po určitú hĺbku, ktorú cyklicky vo funkcií zväčšujem. Ak dosiahne hĺbkovú hranicu, už nevytvorí nasledujúce stavy, ale pôjde postupne na stavy vo fronte. Frontu som riešil ako klasický stack (FIFO). To mi zaručí prehľadávanie do hĺbky z už vytvorených stavov.

Pri riešení som využil funkcie:

**IDF**: hlavná funkcia, v ktorej som cyklicky zväčšoval hĺbku, vytváral nasledujúce stavy(**CreateFront**), kontroloval konečný stav(**ReachGoal**).

**CreateFront**: funkcia v ktorej som vytváral všetky možné pohyby z aktuálneho stavu. Každý pohyb ktorý nie je duplikát(**checkDub**) a nie je blokovaný iným vozidlom(**checkcollision**) je vložený do fronty.

**Opis riešenia v krokoch:**

1. Chcem vybrať stav z fronty. Ak už nie je žiaden stav vo fronte tak idem na krok 5. Inak vyberiem stav a pokračuj na krok 2.
2. Zistím či to nie je finálny stav. Ak je tak končím úspešne. Ak nie tak pokračujem na krok 3.
3. Kontrolujem v akej je stav hĺbke. Ak je na hraničnej hĺbke tak nevytváram z neho ďalšie stavy a skáčem na krok 1. Ak nie je na hraničnej hĺbke tak skáčem na krok 4.
4. Vytváram všetky nasledujúce možné stavy. Pri vytváraní odstraňujem duplikátne stavy. Ak vznikne duplikátny stav, ale je v nižšej hĺbke ako jeho duplikát, tak ho nahradím, inak vyhadzujem tento stav. Po vytvorení všetkých možných stavov pokračujem na krok 1.
5. Ak som prehľadal všetky uzly po danú hĺbku a nenašiel som cielový stav, zvýšim hĺbku o 1.
6. Kontrolujem v akej hĺbke bol najhlbší stav. Ak bol v hĺbke menšej ako maximum, tak už neexistujú stavy hlbšie a nenašiel som cieľ. Končím neúspechom. Ak bol na hĺbkovej hranici tak dám do fronty prvý stav a skočím na 1.

**Opis objektov:**

CarDetail:

Tento objekt obsahuje informácie o vozidle. Jeho orientáciu, farbu, dĺžku, X pozícia, Y pozícia

Node:

Tento objekt obsahuje informácie o stave. Obsahuje všetky pozície vozidiel. Tie sú uložené ako pole objektov CarDetail. Taktiež sa tam nachádza odkaz na rodičovský node, hĺbka node, pohyb ktorý sa uskutočnil na vytvorenie tohto stavu.

**Opis algoritmu:**

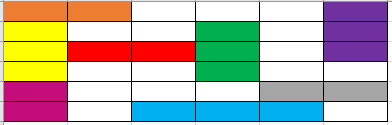
Pri riešení tohto zadania som využil cyklicky sa prehlbujúci algoritmus. Algoritmus funguje na princípe hľadania do hĺbky, ale vždy sa zastaví na stanovenej hĺbke. Hĺbku zväčšujem cyklicky vo funkcií pokiaľ nenájdem finálny stav, alebo som už vygeneroval všetky možné stavy. Týmto algoritmom nájdem vždy optimálne riešenie, takže dokážem určiť že je **úplný**. Časovo je tento algoritmus citeľnejšie pomalší ako prehľadávanie do hĺbky alebo do šírky, ale jeho výhodou je pamäťová nenáročnosť, keďže si pamätá len uzly z danej vetvy.

**Testovanie**

Pri testovaní som si vytvoril niekoľko testovacích scenárov (máp), kde som sa snažil zistiť rýchlosť a spoľahlivosť môjho algoritmu. Vytvoril som niekoľko scenárov kde neexistuje riešenie, niekoľko scenárov s množstvom krokov a niekoľko krátkych scenárov.

Mapy môžem čítať z textového súboru, ale podmienka je aby boli zadané v správnom tvare, a taktiež ich môžem zadávať z konzoly pri spustení.

Ukažkový test:



Počet krokov: 8

Počet spracovaných uzlov: 10596

Kroky:

VLAVO(sive,3)

DOLE(tmavomodre,3)

VPRAVO(oranzove,4)

HORE(zlte,1)

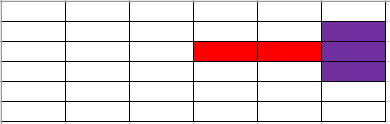
HORE(fialove,1)

VLAVO(svetlomodre,2)

DOLE(zelene,2)

VPRAVO(cervene,3)

1. Test:



Počet krokov: 2

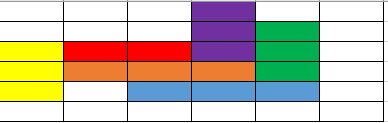
Počet spracovaných uzlov: 17

Kroky:

DOLE(fialove,2)

VPRAVO(cervene,1)

1. Test



Počet krokov: 6

Počet spracovaných uzlov: 753

Kroky:

HORE ( zlte , 2 )

VLAVO ( svetlomodre , 2 )

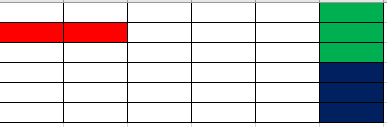
DOLE ( zelene , 2 )

VLAVO ( oranzove , 1 )

DOLE ( fialove , 3 )

VPRAVO ( cervene , 3 )

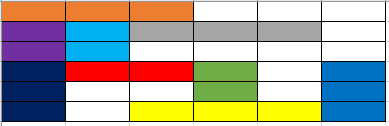
1. Test



Počet krokov: Riešenie neexistuje

Počet spracovaných uzlov: 12

4.Test



Počet krokov: 7

Počet spracovaných uzlov: 3628

Kroky:

VPRAVO ( orandzove , 2 )

HORE ( fialove , 1 )

HORE ( modre , 3 )

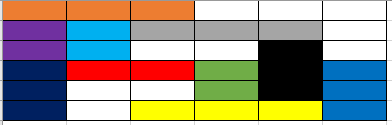
HORE ( tmavomodre , 1 )

VLAVO ( zlte , 2 )

DOLE ( zelene , 1 )

VPRAVO ( cervene , 3 )

5. Test



Počet krokov: 11

Počet spracovaných uzlov: 20175

Kroky:

VPRAVO ( orandzove , 1 )

HORE ( fialove , 1 )

HORE ( modre , 3 )

HORE ( tmavomodre , 1 )

VLAVO ( zlte , 2 )

DOLE ( zelene , 1 )

VPRAVO ( cervene , 1 )

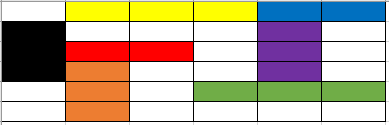
DOLE ( nebove , 2 )

VLAVO ( sede , 1 )

HORE ( cierne , 2 )

VPRAVO ( cervene , 2 )

6. Test



Počet krokov: Riešenie neexistuje

Počet spracovaných uzlov: 2828

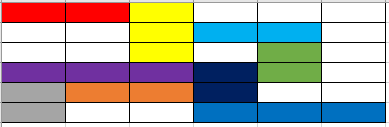
7. Test



Počet krokov: Riešenie neexistuje

Počet spracovaných uzlov: 949

8. Test



Počet krokov: 7

Počet spracovaných uzlov: 1916

Kroky:

VPRAVO ( nebove , 1 )

HORE ( tmavomodre , 2 )

VLAVO ( modre , 2 )

DOLE ( zelene , 2 )

VPRAVO ( fialove , 3 )

DOLE ( zlte , 1 )

VPRAVO ( cervene , 4 )

**Zhodnotenie:**

Cyklicky sa prehlbujúce vyhľadávanie(IDS) spája vyhľadávanie do hĺbky, ktoré vždy nenájde najefektívnejšie riešenie s prehľadávaním do šírky. Požitý algoritmus nájde vždy najkratšiu cestu, aj keď musí veľa krát prejsť tie isté stavy, stromy, keďže je vždy limitovaný do nejakej hĺbky. Výhodou tohto algoritmu je nízka pamäťová náročnosť, keďže si pamätám len uzly aktuálnej vetvy. Ja som pridal do algoritmu rozpoznávanie duplikátov, pretože bez toho by vyhľadanie cieľového stavu trvalo neskutočne dlho, keďže by stále prechádzal duplicitné stavy. Tieto stavy mám uložené v HashMape, ktorá má v sebe uložené len INT hodnoty hĺbky, v ktorej bol daný stav vytvorený. Ak v danej vetve vytvorím stav, ktorý je už v HashMape , tak skontrolujem podľa kľúča, v akej hĺbke bol vytvorený. Ak bol vytvorený v nižšej hĺbke, tak môžem tento stav použiť a prepíšem INT hodnotu hĺbky v HashMape. Podľa môjho názoru je toto jednoduché a efektívne riešenie pre duplikáty. Celkovo je algoritmus časovo o dosť efektívnejší, pri použití nízkeho počtu pomocnej pamäte.

Tento algoritmus je pomalší ako prehľadávanie do hĺbky, pretože musí prehľadávať rovnakú cestu viac krát. Na rozdiel od prehľadávania do hĺbky vždy nájde najefektívnejšiu cestu a nebude trvať nekonečne dlho.

Tento algoritmus je pomalší ako prehľadávanie do šírky, pretože prechádza rovnaké stavy veľa krát, lenže na rozdiel od prehľadávania do šírky je pamäťovo efektívnejší, pretože si pamätám len uzly z danej vetvy.