

Bláznivá križovatka

Škola: Slovenská Technická Univerzita v Bratislave

Fakulta: Fakulta informatiky a informačných technológií

Predmet: Umelá inteligencia

Študent: Bc. František Gič

Cvičiaci: Ing. Ivan Kapustík

Zadanie

Úlohou je nájsť riešenie hlavolamu *Bláznivá križovatka*. Hlavolam je reprezentovaný mriežkou, ktorá má rozmery 6 krát 6 políček 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 políček, môže sa vozidlo pohnúť o 1 až n políček 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.

Implementácia

K vypracovaniu tohto zadania boli použité algoritmy neinformovaného prehľadávania stavového priestoru Depth-first search a Breadth-first search - konkrétne bez použitia rekúzie.

Programoval som v jazyku *JavaScript*, lokálnom environmente - *Node.js* s použitím supersetu *Typescript* pre striktné otypovanie.

Inštalácia

Prerekvizity:

- [Node.js \(https://nodejs.org\)](https://nodejs.org)
- [Node package manager \(https://npmjs.com\)](https://npmjs.com)

V root adresári spustíte nasledovné príkazy:

```
npm install
```

A následne, pre každú transpiláciu typescriptového kódu na javascript a spustenie kódu v node.js:

```
npm run dev
```

Algoritmus

Základný scenár pre prehľadávanie stavov znie nasledovne a jednotlivé body si vysvetlíme na konkrétnej ukážke kódu:

1. Vytvor počiatočný uzol a umiestni medzi vytvorené a zatiaľ nespracované uzly

```
const queue: Crossroad[] = [];  
const visitedStates: StateMap = {};  
queue.push(crossroad);
```

Na obrázku máme možnosť vidieť - *crossroad* je parameter algoritmu, je to počiatočný stav križovatky a umiestňujeme ho do *queue* - náš stack nespracovaných uzlov

2. Ak neexistuje žiadny vytvorený a zatiaľ nespracovaný uzol, skonči s neúspechom – riešenie neexistuje.

Táto časť algoritmu je reprezentovaná klasickým while cyklom.

```
while(queue.length) {  
  ...  
}
```

3. Vyber najvhodnejší uzol z vytvorených a zatiaľ nespracovaných, označ ho aktuálny
 - Ak tento uzol predstavuje cieľový stav, skonči s úspechom – vypíš riešenie
 - Vytvor nasledovníkov aktuálneho uzla a zarad' ho medzi spracované uzly
 - Vytvrd' nasledovníkov a ulož ich medzi vytvorené a zatiaľ nespracované

```

while (queue.length !== 0) {
  const state = queue.shift();
  state.vehicles.forEach((vehicle: Vehicle) => {
    directions.forEach((direction: DIRECTION) => {
      const movedState = state.move(direction, state, vehicle, { steps: 1 });
      if (!movedState) return;

      const hash = md5(JSON.stringify(movedState));
      if (visitedStates[hash]) return;

      visitedStates[hash] = toStateObject(state, movedState, direction, vehicle);
      if (vehicle.id === targetCar.id && movedState.vehicleExists(vehicle.id)) {
        printOperators(visitedStates, md5(JSON.stringify(crossroad)), hash);
        process.exit();
      }

      algorithm === 'dfs' ? queue.unshift(movedState) : queue.push(movedState);
    });
  });
}

console.log(chalk.red('Hlavo!am nema riesenie.'));

```

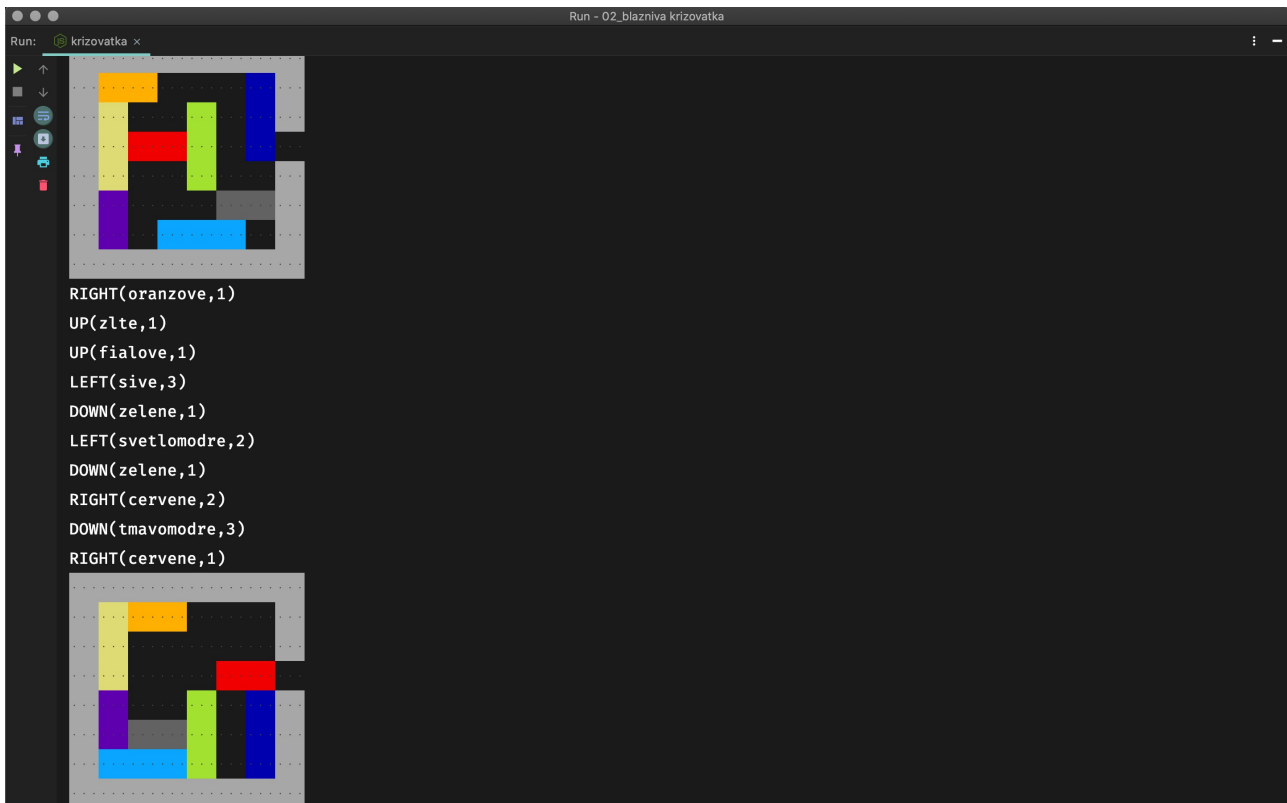
Na obrázku máme možnosť vidieť kompletný algoritmus. Pokúsim sa ho teda opísať:

- Z *queue* uzlov vyberáme prvý. (Nemáme heuristiku, ideme za radom.)
- Pre každé z vozidiel *vehicles* vyskúšame všetky možné smery *directions*.
- Pokiaľ je možné vozidlom pohnúť daným smerom, vznikne nám nový stav (uzol) - *movedState*
- Následne uzol so stavom vkladáme do hashmapy *visitedStates*. Pokiaľ tam už existoval predtým, daný stav neanalyzujeme.
- Skontrolujeme či v danom stave nie je vozidlo práve naše cieľové (algoritmus je abstraktný, cieľovým vozidlom môže byť hociktoré z áut)
- Ak nie, daný stav vložíme:
 - Na koniec zoznamu - *queue* - v prípade *BFS*
 - Na začiatok zoznamu - v prípade *DFS*

Riešenie

Zadanie som spracoval modulárne pre všetky možné vstupy. Jednotlivé vozidlá sú nastaviteľné, majú nastaviteľné poradie, mená, farebnú reprezentáciu, ako aj ich umiestnenie - čo sa mi veľmi osvedčilo pri generovaní testovacích súborov - stačilo zmeniť súbor so súradnicami a je možné vygenerovať N testov.

Modulárna je taktiež $M \times N$ veľkosť križovatky a V vozidiel, D dĺžok a taktiež auta, ktoré je našim cieľom - stačí napísať jeho meno - napr. *cervene*. Východ z križovatky - *exit* sa tak nastaví na pravý okraj križovatky v leveli daného auta.



Za povšimnutie taktiež stojí grafický mód výstupu - okrem stavových reprezentácií, počtu cieľových operátorov ako aj celkový počet uzlov reprezentujem križovatku aj *graficky* v konzole.

- V štandardnom móde sa do konzoly vykreslí počiatočný a koncový stav.
- V grafickom móde sa vykreslí priebežný stav po každom z operátorov (výsledných - správneho riešenia) - čo ma síce núti pamätáť si i stavy, nielen zoznam uzlov a operátorov-
Pre zapnutie grafického módu treba v script.ts nastaviť `process.env.PRINT_GRAPHS = '1'`;

Reprezentácia údajov problému

Stav

V mojom riešení predstavuje stav samotná trieda *Crossroad*. Predstavuje objekt obsahujúci dvojrozmerné pole križovatky, veľkosť (keďže je modulárna). Obsahuje taktiež pole vozidiel.

Vozidlo

Údaje o samotnom vozidle vyzerajú nasledovne:

```
export interface Vehicle {  
  id: number;  
  name: string;  
  color: (text: string) => string;  
  length: number;  
  polarity: 'H' | 'V';  
  position: number[];  
}
```

- vozidlo je v dvojrozmernom poli križovatky reprezentované číslom, *id*.
- pre ľudsku reprezentáciu však vo výstupe operatorov držíme aj názov vozidla pod stringom - *name*.
- tretie pole je pre samotný algoritmus nepodstatný, iba pre grafickú reprezentáciu - hovorí akou farbou má byť vykreslené v konzole.
- moje vozidlá majú modulárnu dĺžku, nie len 2 a 3 - *length*.
- polarita
- pozícia - pole dvoch čísel [i,j]

Operátory

Operátor reprezentuje v každom z uzlov akciu ktorá sa vykonala, aby prišlo k danému stavu. Obsahuje teda

- smer, ktorým sa pohybovala (RIGHT, LEFT, UP, DOWN)
- vozidlo (v skrátenej forme, iba id a názov)
- počet krokov (koľko krokov sa daným smerom pohybovalo)

```
export interface Operator {  
  direction: string;  
  vehicle: {  
    id: number;  
    name: string;  
  };  
  steps: number;  
}
```

Uzol

U nás je daný stav križovatky reprezentovaný triedou *Crossroad*, týmpádom je troška mylné pomenovať uzol ako State.

Obsahuje však :

- *previousHash* - keďže JavaScript nemá smerníky, referencujeme sa do hashmapy pomocou kľúča - ktorým je jedinečný hash na predchádzajúci stav
- *currentState* - samotný stav ktorý uzol reprezentuje
- *operator* - akcia, operátor ktorý sa vykonal aby sa prešlo z predchádzajúceho na aktuálny

```
export interface State {  
  previousHash: string;  
  currentState: Crossroad;  
  operator: Operator;  
}
```

Testovanie

V testovaní som vykonal 6 rôznych testov, v ktorých som menil polohy vozidiel, ako aj poradie operátorov. Následne som porovnával počet celkových uzlov, počet reálnych krokov.

Test 1 (zadanie)

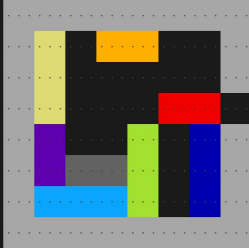
Počiatočný stav: ako v zadaní

Cieľové vozidlo: ako v zadaní

Smery (poradie): UP, RIGHT, DOWN, LEFT


DFS

```
RIGHT(cervene,1)
LEFT(oranžove,1)
UP(tmavomodre,3)
LEFT(oranžove,1)
RIGHT(cervene,1)
DOWN(tmavomodre,3)
RIGHT(cervene,1)
```



```
Number of solution steps: 104
Number of states (total): 545
```

BFS



```
RIGHT(oranžove,1)
UP(zlte,1)
UP(fialove,1)
LEFT(sive,3)
DOWN(zelene,1)
LEFT(svetlomodre,2)
DOWN(zelene,1)
RIGHT(cervene,2)
DOWN(tmavomodre,3)
RIGHT(cervene,1)
```



```
Number of solution steps: 10
Number of states (total): 1072
```

Vidíme, že v prípade Depth-first searchu prehľadal daný algoritmu síce menej jednotlivých uzlov - stavov, avšak výsledná sekvencia operátorov je príliš dlhá na spracovanie. Opačne, Breadth-first search prešiel množstvo rôznych stavov, ale prvá cieľová sekvencia ktorú našiel bola dlhá len 10 krokov (operátorov) !

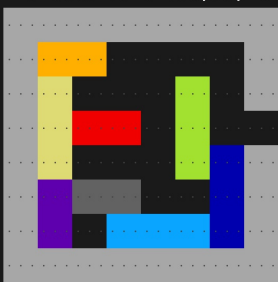
Test 2

Poznámka: Pre debugovacie potreby sme upravili výpis programu - grafické časti, ako aj výpis operandov pokiaľ je dlhší

V tomto teste sme len zmenili horizontálnu pozíciu zeleného a horizontálnu tmavomodrého nákladiaku.

DFS

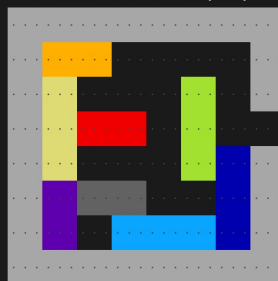
```
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[2,1],"fialove":[5,1],"sive":[5,2],"zelene":[2,5],
"svetlomodre":[6,3],"tmavomodre":[4,6]}



Cielove vozidlo: cervene
Smery: [ 0, 1, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
Number of solution steps: 450
Number of states (total): 2341
```

BFS

```
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[2,1],"fialove":[5,1],"sive":[5,2],"zelene":[2,5],
"svetlomodre":[6,3],"tmavomodre":[4,6]}



Cielove vozidlo: cervene
Smery: [ 0, 1, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
RIGHT(cervene,1)
DOWN(zelene,1)
LEFT(svetlomodre,1)
DOWN(zelene,1)
RIGHT(cervene,2)
Number of solution steps: 5
Number of states (total): 581
```

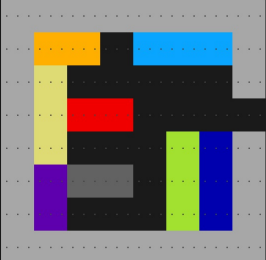
Vidíme že v tomto teste sa DFS veľmi potrápil - a to ide len o tri kroky ktoré človek vidí na prvý pohľad.

Chápeme však, čo je za tým. Dôležitým faktorom v oboch algoritmoch je poradie áut, i poradie operátorov. V tomto teste prešiel skoro všetky ostatné vozidlá až kým sa dostal k zelenému - čo vytvára veľké množstvo stavov.

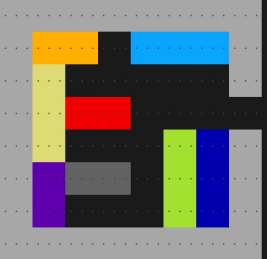
Test 3

Schválne sme vybrali nastavenie, ktoré je jednoznačné. Nastavili sme ako prvý smer doprava, a vozidlá sme upravili tak, aby cieľové vozidlo malo voľnú cestu. Predpokladali sme, že BFS totálne zvalcuje DFS a tak sa aj stalo. Týmto sme potvrdili hypotézu o vplyve poradia smerov na počet celkových uzlov algoritmu.

DFS

```
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[2,1],"fialove":[5,1],"sive":[5,2],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: cervene
Smery: [ 1, 0, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
Number of solution steps: 44
Number of states (total): 242
```

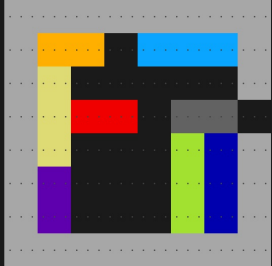
BFS

```
krizovatka x
Run - 02_blazniva_krizovatka
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[2,1],"fialove":[5,1],"sive":[5,2],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: cervene
Smery: [ 1, 0, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
RIGHT(cervene,3)
Number of solution steps: 1
Number of states (total): 26
```

Test 4

Schválňny test, na otestovanie správnosti implementácie algoritmov.

DFS

```
krizovatka x
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[2,1],"fialove":[5,1],"sive":[3,5],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: cervene
Smery: [ 0, 1, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
Hlavoľam nema riesenie.
```

BFS

```
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[2,1],"fialove":[5,1],"sive":[3,5],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: cervene
Smery: [ 0, 1, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
Hlavoľam nema riesenie.
```

Test 5

Pre demonštráciu modularity cieľového vozidla sme vytvorili test s blokadou sivého autíčka a nastavním sivého ako nášho cieľového vozidla.

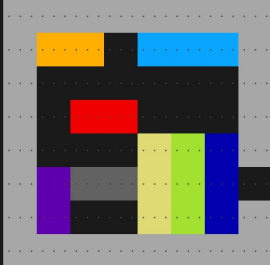
DFS

```

/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"

Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[4,4],"fialove":[5,1],"sive":[5,2],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

```

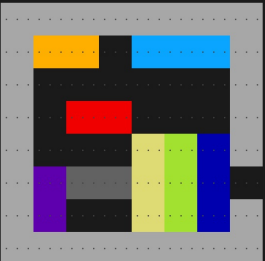


```

Cielove vozidlo: sive
Smery: [ 0, 1, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
Number of solution steps: 80
Number of states (total): 471

```

BFS

```
krizovatka x
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[4,4],"fialove":[5,1],"sive":[5,2],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: sive
Smery: [ 0, 1, 2, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
UP(zlte,2)
RIGHT(sive,1)
UP(zelene,2)
RIGHT(sive,1)
UP(tmavomodre,2)
RIGHT(sive,1)
Number of solution steps: 6
Number of states (total): 1260
```

Test 6

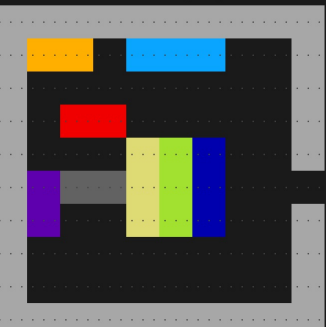
Posledným krokom je demonštrácia modularity samotnej križovatky. A taktiež demonštrácia, že BFS (aj keď to z niektorých našich testov vyzeralo) nemusí byť vždy lepší. Všetko totiž závisí od daného vstupu, a preto sme vytvorili jeden nasledovný. Zväčšili sme plán na 8x8, cieľové autíčko je sivé a zablokovali sme ho. V prípade DFS - prvý hit, síce 626 krokov, ale málo (6k) uzlov. Tento setup však bol oriešok pre BFS a trval asi najviac zo všetkých - okolo jednej minúty (Quad core i5 2.4GhZ, 16GB RAM). 55 tisíc uzlov!

DFS

```
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[4,4],"fialove":[5,1],"sive":[5,2],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: sive
Smery: [ 1, 2, 0, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
Number of solution steps: 626
Number of states (total): 6314
```

BFS

```
/usr/local/bin/node "/Users/feri/Library/Mobile Documents/com~apple~CloudDocs/STU/1.Ing/LS/UI/zadania/02_blazniva
krizovatka/dist/script.js"
Pociatocny stav: {"cervene":[3,2],"oranzove":[1,1],"zlte":[4,4],"fialove":[5,1],"sive":[5,2],"zelene":[4,5],
"svetlomodre":[1,4],"tmavomodre":[4,6]}

Cielove vozidlo: sive
Smery: [ 1, 2, 0, 3 ] (0 - UP, 1 - RIGHT, 2 - DOWN, 3 - LEFT)
DOWN(zlte,2)
RIGHT(sive,1)
DOWN(zelene,2)
RIGHT(sive,1)
DOWN(tmavomodre,2)
RIGHT(sive,3)
Number of solution steps: 6
Number of states (total): 54816
```

Záver

Väčšinu problémov som vysvetlil v samotnom testovaní. Z neho vyplýva nasledovné: Úspešnosť neinformovaných prehľadávaní bez akéhokoľvek ohodnotenia daných stavov je síce v niektorých prípadoch výhodné, ale veľmi závisí od daného nastavenia - vstupu, pozície autíčok, poradiu operátorov - smerov. V mojom zadaní by som vylepšil napríklad počet krokov - momentálne sa autíčka pohybujú v jednej operácii o jeden krok, i keď je to pripravené na modulárny pohyb (stačí zmeniť parameter funkcie). Bolo by zaujímave sledovať napríklad správanie pri maximálnom možnom pohybe - pohybovať vozidlo, kým nenarazí.