Andrejčák Daniel, ID: 120746
Streda 13:00

UI- zadanie 1

CPU: AMD Ryzen 7 4700U

RAM: 16,0 GB

OS: Windows 10

IDE: Visual Studio (C++ 14)

Kompilácia

- súbor sa skladá z main.cpp a structures.h
- pre g++: g++ main.cpp -o main

Spustenie programu

- pre správne spustenie programu musí používateľ zadať CLI argumenty
  1. argument je číslo scenára (1 až 6)
  2. argument je typ prehľadávacieho algoritmu (DFS, BFS)
  3. voliteľný argument je maximálny požadovaný počet ťahov – ak používateľ tento argument nezadá, použije sa predvolený maximálny počet ťahov (65535)

Riešený problém

Problém 1, úloha A

Použite algoritmus prehľadávania do šírky a do hĺbky. Porovnajte ich výsledky.

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.

Andrejčák Daniel, ID: 120746
Streda 13:00

structures.h

Tento hlavičkový súbor obsahuje štruktúry potrebné na implementáciu prehľadávacích algoritmov na daný problém. Ďalej obsahuje štruktúry a funkcie umožňujúce prehľadný a čitateľný výpis programu pre užívateľa. Obsahuje taktiež funkcie na hashovanie a porovnanie uzlov

Štruktúra reprezentujúca uzol

```
class Node {
public:

    std::vector<Car> cars;

    Node* pNode = nullptr;

    Color color = Color::NULLCOLOR;
    char dir = 0;
    unsigned short n = 0;

    unsigned short depth = 0;

```

Každý uzol v stavovom priestore obsahuje:

- stav (hracia plocha s určitým usporiadaním aut)
- orientovanú hranu (ukazovateľ na predchádzajúci uzol)
- farbu auta, ktoré sa pohlo
- smer, v ktorom sa pohlo
- počet políčok o ktoré sa pohlo
- hĺbku v akej sa v grafe – stavovom priestore nachádza

Štruktúra reprezentujúca auto

```
class Car {
public:

    Color color = Color::NULLCOLOR;
    unsigned short xAxis;
    unsigned short yAxis;
    unsigned short size;
    char dir;

```

Každé auto v stave obsahuje:

- farbu auta
- x a y pozíciu auta (**! OD 0 do 5 !)**
- veľkosť auta (2 alebo 3, tak ako sa písalo v zadaní)
- orientáciu (horizontálne, alebo vertikálne)

Andrejčák Daniel, ID: 120746
Streda 13:00

main.cpp

Súbor main.cpp je hlavný program, ktorý obsahuje funkcie na vytvorenie stavového priestoru, prehľadávacie algoritmy na nájdenie finálneho stavu a niekoľko pomocných funkcii, ktoré slúžia na inicializáciu počiatočného stavu, vykonanie ťahov na hracej ploche a na výpis použitých ťahov a finálneho stavu.

searchAlgorithm()

```
Node* searchAlgorithm(Node* root)
```

Funkcia searchAlgorithm() je hlavná funkcia, ktorá vykonáva DFS alebo BFS. Vstupným parametrom je počiatočný stav – uzol.

Ako prvé sa vo funkcii skontroluje, či červené auto nie je blokované iným horizontálne otočeným autom. Ak áno, to by znamenalo, že hra sa nikdy neskončí a zbytočne by prebehol prehľadávaci algoritmus a preto program skončí s výpisom, že červené auto sa nikdy nedostane z parkoviska.

Ako druhé sa skontroluje, či sa nedá hra ukončiť už z počiatočného stavu, ak áno funkcia moveRedToFinal() vytvorí konečný stav, kde bude červené auto na pozícii (4,2), ktorá sa chápe ako že červené auto vyšlo z parkoviska. Funkcia vytvorí orientovanú hranu medzi počiatočným a konečným stavom a program prejde na výpis.

Ak sa z počiatočného stavu ešte nedá hra ukončiť, tak uzol, ktorý reprezentuje počiatočný stav (Node* root) pridá do vectora nodesToProcess, ktorý slúži ako dátová štruktúra stack / queue potrebná pre algoritmy DFS a BFS. Ďalej začne prebiehať samotný prehľadávaci algoritmus.

```
while (!nodesToProcess.empty()) {
    node = pop(nodesToProcess);

    if (node->depth >= MAXDEPTH)
    {
        delete node;
        continue;
    }

    for (auto& car : node->cars)
    {
        //kazdy pohyb
        if (car.dir == 'v')
        {
            finalNode = move(node, car, &up, &moveUp);
            if (finalNode) { break; }

            finalNode = move(node, car, &down, &moveDown);
            if (finalNode) { break; }
        }
        else
        {
            move(node, car, &left, &moveLeft);

            move(node, car, &right, &moveRight);
        }
    }

    visited.insert(node);

    if (finalNode)
    {
        visited.insert(node);
        visited.insert(finalNode);

        return reversePath(finalNode);
    }
}
```

Andrejčák Daniel, ID: 120746
Streda 13:00

Samotný algoritmus funguje tak, že zo stacku / queue nodesToProcess vyberie uzol, ktorý je na rade na „preskúmanie“ a ako prvé skontroluje jeho hĺbku v grafe, či nepresiahla určitú úroveň, ktorá v podstate určuje maximálny požadovaný počet ťahov (môže byť nastavená používateľom). Ak uzol nepresiahol maximálnu hĺbku v grafe, tak sa začne jeho preskúmavanie. To znamená že pre daný stav vyskúša pohnúť so všetkými autami v rôznych smeroch o 1 až n políčok (n je v tomto prípade 4). To zabezpečuje funkcia move(), ktorá generuje nové stavy – uzly a vkladá ich do nodesToProcess.

Pri pohyboch aut, ktoré sú otočené vertikálne môže nastať situácia, že sa červenému autu uvoľní cesta z parkoviska. Preto po pohyboch s autami otočenými vertikálne sa skontroluje, či takýto stav nenastal.

Ak áno, tak algoritmus skončí prehľadávanie a funkcia vráti počiatočný uzol s cestou k finálnemu uzlu (to zabezpečí funkcia reversePath(), ktorá obráti cestu z finálneho uzlu k počiatočnému uzlu).

Ak nie, tak skončí prehľadávanie tohto uzla a vloží sa do poľa visited, ktoré slúži na to, aby sa pri generácií nových stavov znova negenerovali už existujúce stavy.

move()

```
1  Node* move(Node* node, const Car& car, unsigned short (*dirValidFunc), Node* (*dir))
```

Funkcia move slúži na generáciu nových stavov – uzlov. Pri volaní zo searchAlgorithm() dostane ako argumenty, uzol z ktorého ma generovať ďalšie stavy – uzly, auto, ktorým ma pohnúť aby vygeneroval nový stav – uzol a smer v ktorom ma autom pohnúť. Táto funkcia sa pokúsi vykonať pohyb o 1 až n políčok, pričom, ak zistí, že sa nedokáže pohnúť o napr. 2 políčka, tak ďalšie pohyby o viacero políčok skúšať nebude.

Generácia nových stavov funguje tak, že spraví kópiu stavu, ktorý bol poslaný do funkcie ako argument, v ňom reálne spraví posun autom a potom skontroluje, či sa taký stav nenachádza v poli visited. Ak áno, novo vygenerovaný uzol odstráni a pokračuje ďalej v posúvaní auta o viacej políčok.

Ak sa nenachádza, znamená to, že nový stav je unikátny a uzlu sa priradia ostatné atribúty ako farba auta, ktorým bolo posunuté, smer v ktorom sa posunulo, počet políčok o koľko sa posunulo, orientovaná hrana smerujúca na predchádzajúci uzol a hĺbka uzla v grafe.

Potom sa skontroluje, či z novo vzniknutého stavu nemôže vzniknúť finálny stav (červené auto sa má možnosť dostať z parkoviska). Ak áno, vytvorí sa finálny uzol, priradí sa mu ohraničená hraná smerujúca na novo vzniknutý stav a funkcia vráti finálny uzol. Potom nasleduje proces, otočenie cesty od finálneho stavu k počiatočnému stavu, ako bol spomenutý v texte vyššie.

std::vector<Node*> nodesToProcess

Tento vector funguje ako dátové štruktúry stack, alebo queue podľa použitého prehľadavacieho algoritmu.

Pri DFS sa novo vzniknuté uzly ukladajú do stacku, a každá iterácia algoritmu si vyberie z vrchu stacku uzol, ktorý bude prehľadávať. Pri tomto algoritme funguje koniec vectora ako top stacku. Uzly sa ukladajú na koniec vectora a z konca vectora sa zároveň vyberajú na spracovanie. To zabezpečí, že nastane prehľadávanie do hĺbky, pretože vždy sa budú vyberať najneskôr vygenerovane uzly.

Andrejčák Daniel, ID: 120746
Streda 13:00

Pri BFS sa novo vygenerované uzly ukladajú do queue a každá iterácia algoritmu berie zo začiatku queue. Pri tomto algoritme funguje tento vector ako queue. Uzly sa ukladaju na koniec queue a vyberajú sa zo začiatku. To zabezpečí, že nastane prehľadávanie do šírky, pretože sa vždy budú vyberať uzly, ktoré vznikli najskôr.

Keďže stack aj queue je reprezentovaná tým istým vectorom, a vyhľadávací algoritmus sa vyberá cez CLI argument pri spustení, tak až počas runtime programu je určené, ktorá s týchto dátových štruktúr bude použitá. To znamena, že funkcie ako push a pop musia byť implementované pre obidve tieto dátové štruktúry.

Ako už bolo spomenuté, vkladanie prvkov je pri oboch dátových štruktúrach na koniec vectora pomocou funkcie push_back(). Pri stacku sa uzly vyberajú z konca vectora, pri queue sa vyberajú zo začiatku.

Funkcia pop slúži ako ukazovateľ na jednu z týchto dvoch funkcii. Podľa zvoleného algoritmu sa vyberie dátová štruktúra, ktorá sa použije a teda sa aj priradí správna funkcia na vyberanie uzlov z danej dátovej štruktúry

```
Node* (*pop)(std::vector<Node*>&);

Node* popDfs(std::vector<Node*>& vec) {
    Node* node = vec.back();

    vec.pop_back();

    return node;
}

Node* popBfs(std::vector<Node*>& vec) {
    Node* node = vec.front();

    vec.erase(vec.begin());

    return node;
}
```

```
std::string stringArg(argv[2]);

if (stringArg == "DFS")
{
    pop = popDfs;
}
else if(stringArg == "BFS")
{
    pop = popBfs;
}
```

std::unordered_set<Node*, CustomNodeHash, CustomNodeEqual> visited

Tento unordered_set funguje ako pole kde sa uchovávajú uzly, ktoré už boli preskúmané, teda ich potomkovia už sú uložené vo vectore nodesToProcess. Tento set sa používa pre prípad, že algoritmus vygeneruje rovnaký stav, aký už bol niekedy predtým vygenerovaný a došlo by ku vzniku zbytočnej duplicitnej vetvy. Unordered_set je použitý pre to, lebo má konštantný čas vyhľadávania, keďže používa hash table. Ten je definovaný v súbore structures.h v štruktúre CustomNodeHash. Taktiež tam je implementovaná vlastná funkcia na porovnávanie uzlov, ktorá porovnáva iba stavy, ktoré obsahujú, pretože nás zaujíma iba stav a nie ostatné atribúty, ktoré uzol obsahuje, keď chceme zistiť či sa 2 rôzne uzly rovnajú (zaujíma nás iba stav a nie spôsob akým vznikol).

Andrejčák Daniel, ID: 120746
Streda 13:00

Testovacie scenáre

V programe sa nachádza niekoľko scenárov, teda rozložení aut na hracej ploche. Ako už bolo písane v úvode dokumentácie, používateľ si môže jeden z týchto scenárov vybrať a použiť na neho DFS, alebo BFS algoritmus.

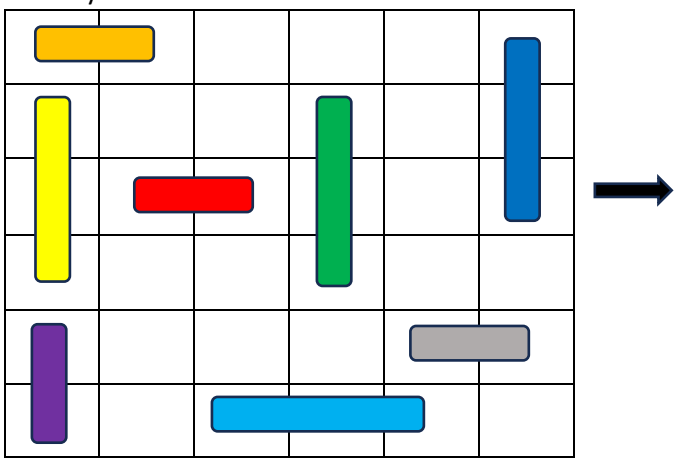
Validný vstup pre funkciu loadCars(), ktorá je zodpovedná za vytvorenie počiatočného uzla, je string v tvare „farba1 x1 y1 velkost1 orientacia1 farba2 x2 y2 velkost2 orientacia2 farba3 x3 ...“

```
scenar = "oranzove 0 0 2 h zlte 0 1 3 v ruzove 0 4 2 v cervene 1 2 2 h zelene 3 1 3 v modre 5 0 3 v sive 4 4 2 h svetlomodre 2 5 3 h";
```

Momentálne však program **nemá možnosť pre používateľský vstup**. Používateľ si však môže vybrať 1 zo 6 scenárov.

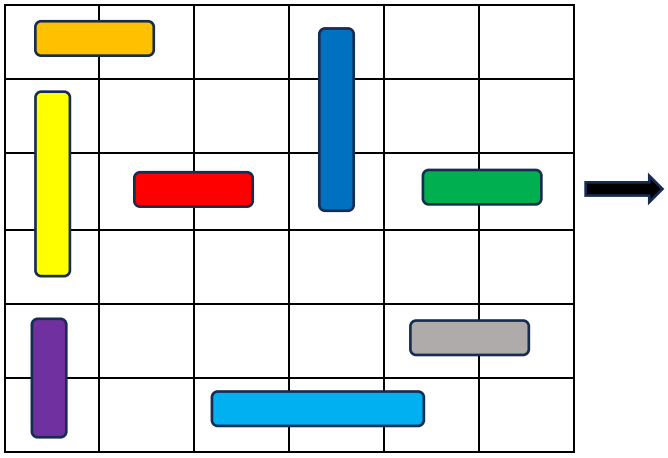
1. scenár

- rovnaký ako v zadaní



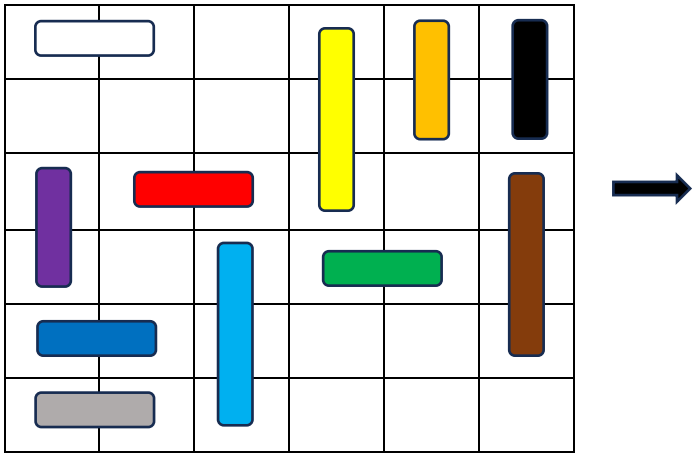
2. Scenár

- Červené auto je blokované horizontálne otočeným autom a teda nikdy nebude môcť vyjsť z parkoviska

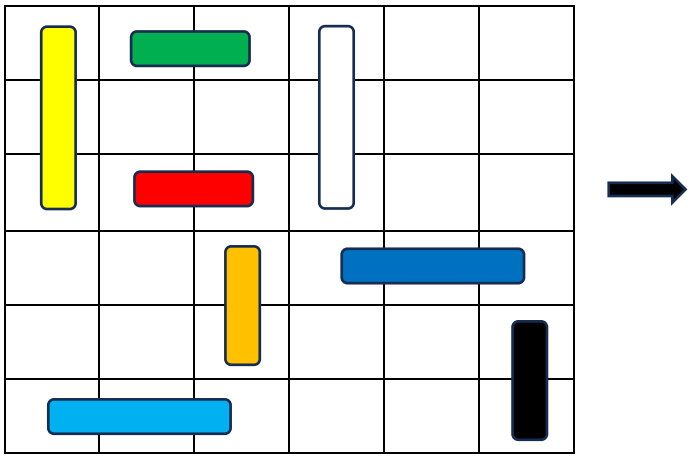


Andrejčák Daniel, ID: 120746
Streda 13:00

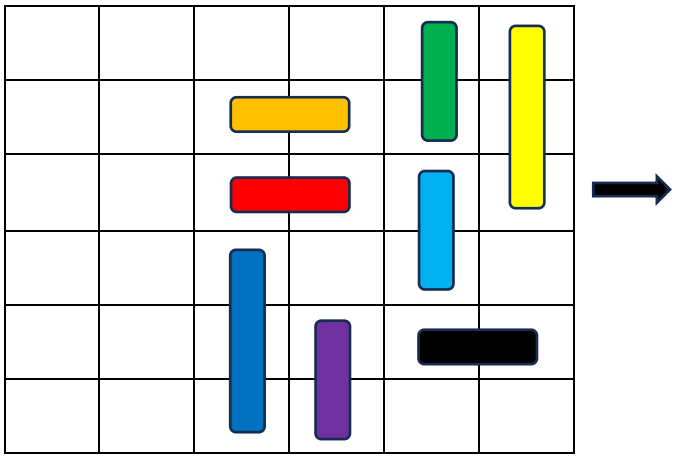
3. Scenár



4. Scenár



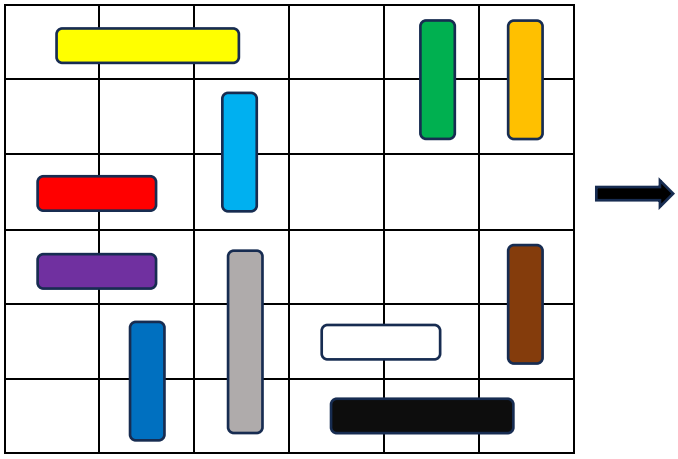
5. Scenár



Andrejčák Daniel, ID: 120746
Streda 13:00

6. Scenár

- neriešiteľné



Záver

Algoritmy DFS a BFS sú všeobecne známe a často využívané algoritmy na prehľadávanie grafov. V tejto úlohe slúžili na prehľadávanie stavového priestoru hry Bláznivá križovatka.

Obidva tieto algoritmy majú svoje výhody aj nevýhody. DFS je všeobecne rýchlejší algoritmus, ktorý zároveň potrebuje menej zdrojov, no nie vždy však nájde optimálne riešenie, teda minimálny počet ťahov. BFS, ktorý bol omnoho náročnejší či už časovo, ale aj pamäťovo vždy zaručil nájdenie optimálneho výsledku, teda najmenší počet ťahov.

Oba tieto algoritmy majú svoje uplatnenie a záleží od špecifických požiadaviek a situácie, aby sa dalo určiť, ktorý z týchto dvoch algoritmov je lepší.