Slovenská Technická Univerzita v Bratislave

Fakulta informatiky a informačných technológií

Popolvár

Zadanie 3

Predmet: Dátové štruktúry a algoritmy

Obdobie: Letný semester 2019/2020

Cvičiaci: Ing. Dominik Macko, PhD.

Študent: Bc. František Gič

Obsah

Implementácia

Použitý algoritmus

K vyriešeniu zadania som použil Dijkstrov algoritmus s minimálnou haldou.
Tento algoritmus je vylepšený algoritmus prehľadávania stavového priestoru v grafe bez nutnosti navštívenia uzlov – prehľadania celého stavového priestoru. Dôležité je v ňom kladné ohodnotenie hrán medzi uzlami. V našom prípade je ohodnotenie hrán označených cestou ‘C’ za 1 jednotku času, a hrany označené ‘H’ – tzv. hustým porastom, ohodnotené 2 jednotkami času.
K pamätaniu si jednotlivých uzlov ktoré má algoritmus navštíviť používam binárnu minimálnu haldu. Halda je výbornou dátovou štruktúrou na riešenie tohto konkrétneho problému práve z dôvodu, že prvky v nej sú usporiadané podľa zvolených pravidiel a celá štruktúra môže byť reprezentovaná ako jednodimenzionálne pole prvkov, kde nasledovníci daného prvku sú umiestnený podľa vzorca `2i` a `2i+1` (v prípade indexovania od 0 `2i+1` a `2i+2`). To znamená, že v prípade vloženia alebo výberu prvku, prvky s menším ohodnotením (v našom prípade cena cesty), “prebublávajú” prvky na vrch štruktúry – či už najväčšie alebo najmenšie prvky, to závisí, či je halda minimálna alebo maximálna.
Vďaka tejto štruktúre Dijkstrov algoritmus tak zakaždým z haldy vyberie prvok s minimálnym ohodnotením a navštívi všetky uzly grafu ktoré s ním priamo susedia, okrem prvkov z ktorých prišiel.

```
// X & Y direction vectors
const int sX[4] = {-1, 0, 1, 0};
const int sY[4] = {0, 1, 0, -1};

Node *actual;
do {
    actual = pop(heap);
    for (int i = 0; i < 4; i++) {
        Node *m = move(actual, map, sX[i], sY[i]);
        if (m == NULL ) continue;
        m->prev = actual;
        push(m, heap);
    }
} while (actual->coordinates.x != stop->x || actual->coordinates.y != stop->y);
```

Dátové štruktúry

V implementácií tohto zadania som použil 4 dátové štruktúry, a to konkrétne `Coordinates`, `Node`, `Heap` *a* `Map`.

`Coordinates` – štruktúra reprezentujúca koordináty v dvojrozmernom priestore – x a y

`Node` – uzol reprezentujúci bod prechodu grafom. Obsahuje vyššie spomenuté koordináty určujúce dané políčko v mape, hodnotu doterajšej cesty od začiatku a smerník na predchádzajúci uzol z dôvodu backtrackingu koordinátov pre výstup.

`Heap` – wrapper minimálnej haldy. V tejto štruktúre je smerník na (smerník na array typu Node) – toto riešenie som použil z dôvodu, že vo viacerých funkciách tento smerník realokujem pri zväčšovaní kapacity haldy, momentálnu veľkosť haldy a maximálnu veľkosť. Posledné z uvedených som zvolil práve z dôvodu, že realokovanie a zväčšovanie haldy sa uskutočňuje v `O(n)` a nie v `O(n^2)`, v prípade, že by som zväčšoval zakaždým pri vložení prvku.

`Map` – wrapper dvojrozmerného poľa znakov. Toto riešenie som zvolil z jediného dôvodu, a to, že som nechcel posielať veľkosti mapy x a y zakaždým ako parametre ☺

```c
typedef struct Coordinates {
    int x;
    int y;
} Coordinates;

typedef struct Node {
    Coordinates coordinates;
    int value;
    struct Node *prev;
} Node;

typedef struct Heap {
    Node **heap;
    int size;
    int max_size;
} Heap;

typedef struct Map {
    char **map;
    int x;
    int y;
} Map;
```

Matica navštívenia

Z dôvodu veľkého množstva možných uzlov v grafe si samozrejme nemôžeme pamätať tie, ktoré sme už navštívili.
Preto som sa rozhodol použiť dvojrozmerné pole (maticu) typu short pre uchovávanie si informácie, či som na danom mieste už bol. Táto informácia je do matice zapísaná v prípade, že dané políčko mapy je vybraté z haldy a aktuálne prehľadávame všetkých jeho potomkov. V prípade, ak je dané políčko mapy znovu nájdené niektorým z jeho susedov v grafe, do haldy sa nevloží.

```
// X & Y direction vectors
const int sX[4] = {-1, 0, 1, 0};
const int sY[4] = {0, 1, 0, -1};
    František Gič, 09/05/2020, 12:03 AM • Implement inserting into minheap
Node *actual;
do {
    actual = pop(heap);
    visited[actual->coordinates.x][actual->coordinates.y] = 1;
    for (int i = 0; i < 4; i++) {
        Node *m = move(actual, map, sX[i], sY[i]);
        if (m == NULL || visited[m->coordinates.x][m->coordinates.y]) continue;
        m->prev = actual;
        push(m, heap);
    }
} while (actual->coordinates.x != stop->x || actual->coordinates.y != stop->y);
```

Pre porovnanie, výpis počtu prvkov haldy medzi štartom a drakom v teste č.1 s maticou navštívenia a bez nej.

```
Zadajte cislo testu (0 ukonci program):
1
heap->size: 35
```

```
Zadajte cislo testu (0 ukonci program):
1
heap->size: 3646
```

Hľadanie princezien

Daným spôsobom sme teda našli cestu s najmenším možným ohodnotením po draka, avšak ako pokračovať, keď máme dynamický počet princezien?
Vybral som si teda spôsob rekurzívneho prehľadávania permutácií poradí princezien.
Vo funkcií `zachran_princezne` sa z mapy načítavajú koordináty princezien na danej mape, takže tieto koordináty sú nám známe. Vytvoríme všetky možné permutácie poradia týchto princezien v funkcií `permutePrincesses` a na každé výsledné pole princezien s rozličným zavoláme rekurzívnu funkciu `getLastPrincess`, ktorá sa postupne vnára až po prvú princeznú v poli, aplikuje naň Dijkstrov algoritmus od draka, potom zavolá algoritmus od koordinátov od prvej princeznej po ďalšiu atď. Výsledkom tejto funkcie je Uzol, s polohou poslednej princeznej, a keďže sme uzly sú definované štruktúrou Node, poznáme ako aj polohu, tak aj ohodnotenie a spájaný zoznam týchto štruktúr.

```
Node *getLastPrincess (Map *map, Node *start, Coordinates *princesses, int n_of_princesses) {
    if (n_of_princesses == 1)
        // First princess from a permutated array
        return dijkstra(map, start, princesses);
    else {
        // Get the princess before
        Node *temp = getLastPrincess(map, start, princesses, n_of_princesses: n_of_princesses - 1);
        // Run dijkstra from princess before till current princess
        return dijkstra(map, temp, stop: princesses + (n_of_princesses - 1));
    }
}
```

Takáto štruktúra sa nám vráti v každej permutácií, a tak nám stačí z týchto výsledkov vybrať jeden, ktorý má najmenšiu „cenu“. Tento spájaný zoznam následne už len pretraverzujeme a zapíšeme koordináty do poľa integerov. (Toto robíme len z dôvodu že chceme v riešení použiť testovaciu funkciu na vyhodnotenie výsledku od cvičiaceho).

Časová a pamäťová zložitosť

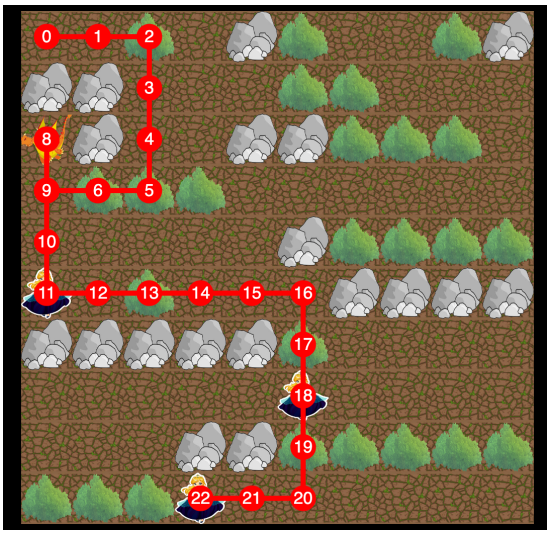
Časová zložitosť vkladania do haldy je $O(log_2\ n)$. Časová zložitosť Dijkstrovho algoritmu je `O(VE log VE)`, kde V je počet vrcholov a E je počtom hrán. Pamäťová zložitosť je `O(n*m + E)`, kde `n` a `m` sú rozmery mapy.

Testovanie

K testovaniu môjho zadania som použil celkom 6 testov, z toho dva boli dané cvičiacim. Testovanie prebehlo v štýle že som vygeneroval náhodné mapy. (nakódil som si JavaScriptový nástroj, v ktorom vyklikám dané štvorčeky, a ono mi to vráti maticu hodnôt ☺) Do takýchto máp som ešte princezné a draka, vložil do súboru a do metódy main ich načítavanie. Následne som výstup z nich overil v grafickom zobrazení na adrese: https://popolvar.surge.sh/, kde bolo zaujímavé sledovať cestu popolvára. Niektoré z ciest by som osobne spravil úplne ináč, napríklad v teste č.4. Taktiež, všimol som si, že smerovanie popolvára závisí pri rovnako ohodnotených hranách čisto od poradia, v akom aplikujem smerové pravidlá.

Test 1

Základný test, tri princezné a drak, špecifikovaný cvičiacim. Zameriava sa hlavne na správnosť riešenia.

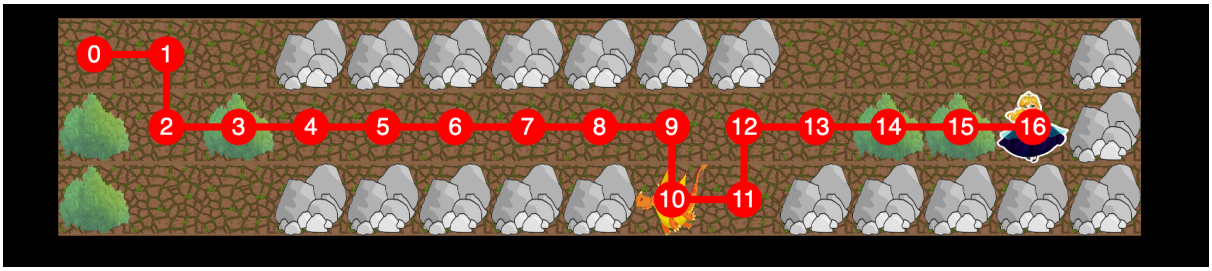


Test 2

Test 2 preskočíme, nakoľko je to totožný test s prvým, avšak načítaný programaticky, nie zo vstupného súboru. Tento test však obsahuje jeden problém. Dvojdimenzionálna mapa sa alokuje dynamicky, v heape pomocou mallocu, avšak samotné polia charov sú naplnené manuálne zápisom mapa[i] = ‘CNCNNCCCD’. Táto hodnota však nie je uložená v heape, ale v stacku, tým pádom funkcia free, ktorá sa volá na jednotlivé dimenzie mimo kontextu switch spadne na uvoľňovaní pamäte ktorý nebol alokovaná v heape ☺

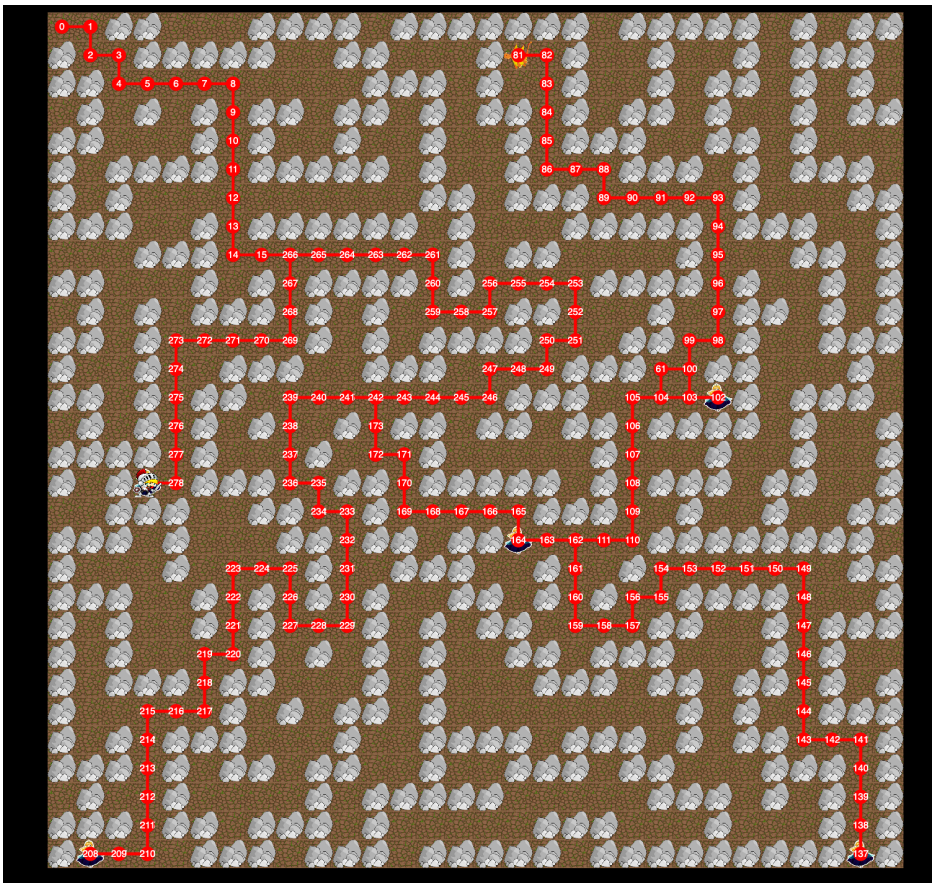
Test 3

Test 3 je zameraný na správnosť implementácie Dijkstrovho na ohodnotených hranách. Vidíme, že algoritmus si nevybral cestu ktorá je voľnejšia, ale prešiel cez kríky (hustý porast), i keď sú dané cesty lokálne „drahšie“, obchádzka „lacnejšími“ cestami by bola v konečnom dôsledku „drahšia“.



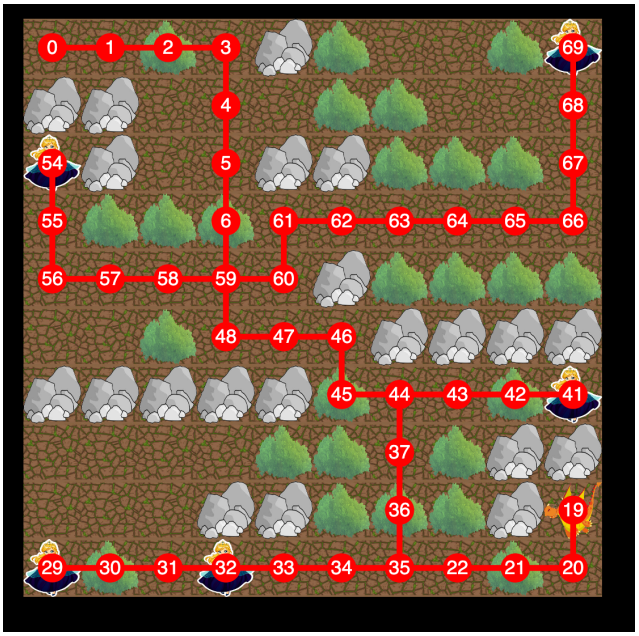
Test 4

Je zameraný na veľkú mapu s maximálnym počtom princezien. Vytvoril som bludisko v mojom generátore a umiestnil som draka a princezné. Bolo zaujímavé pozorovať niektoré rozhodnutia popolvára, ale neskôr som si uvedomil že postupoval správne.



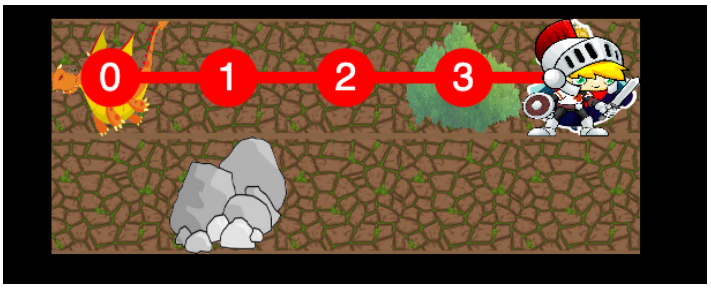
Test 5

Zameraný na umiestnenie draka na absolútny protipól štartu popolvára a umiestnenia maximálny počet princezien v čo najväčšom rozpätí v rovnakom bludisku akým je bludisko č.1.



Test 6

Posledný test, správnosť implementácie na bludisku kde drak je zároveň na štartovnej pozícií a s jednou princeznou.



Zhodnotenie

Dané cvičenie mi pomohlo lepšie pochopiť Dijkstrov algoritmus, ako vyhľadávať v grafoch bez nutnosti prehľadať všetky stavy. Taktiež som sa naučil, čo je binárna halda, a ako sa implementuje, nakoľko som s touto štruktúrou ešte neprišiel do kontaktu pred týmto zadaním. Zistil som, že ukladanie prvkov a „prebublávanie“ je úžasná vec ☺

Ospravedlňujem sa za zmiešanie anglických a slovenských premenných a názvov funkcií a procedúr v kóde, čo môže vyvolať zmätok pri čítaní kódu – primárne programujem v angličtine, ale nevedel som, či bude daná funkcionalita testovaná bez môjho main-u a či môžem meniť hlavnú procedúru `zachran_princezne`, a tak som tieto funkcie použil, tak ako boli v príklade doručeným od cvičiaceho a ostatný, môj kód programoval v angličtine.

Referencie

BARI, A. *Dijkstra's Algorithm - Single Source Shortest Path - Greedy Method.* [cit.10.05.2020]. Dostupné na internete: https://www.youtube.com/watch?v=XB4MIexjvY0

COMPUTERPHILE. Dijkstra’s Algorithm. [cit. 10.05.2020]. Dostupné na internete: https://www.youtube.com/watch?v=GazC3A4OQTE

KOREŠPONDENČNÝ SEMINÁR Z PROGRAMOVANIA. *Dijkstrov algoritmus*. [cit. 10.05.2020]. Dostupné na internete: https://www.ksp.sk/kucharka/dijkstra/

KOREŠPONDENČNÝ SEMINÁR Z PROGRAMOVANIA. *Halda.* [cit. 10.05.2020]. Dostupné na internete: https://www.ksp.sk/kucharka/halda/