Zadanie 3 - Popolvár

Filip Vida, Fakulta informatiky a informačných technológií STU v Bratislave 9/05/2020

Binárna Halda

Binárna halda je dátová štruktúra a typ binárneho stromu. Je to jednoduchý typ binárneho stromu a má 2 obmedzenia. Dĺžka vetiev sa líši maximálne o 1 a podľa typu haldy je nutné aby prvky pod aktuálnym prvkom boli menšie (max-heap), alebo väčšie(min-heap).Prvá podmienka je ľahko dosiahnuteľná, keďže si binárnu haldu reprezentujeme pomocou poľa. Toto znamená že prvok naľavo máme na indexe 2*x, a pravý na indexe 2*x+1. Pri vykonávaní rôznych operácií na heape treba heap následne upraviť. Pri odstránení rootu treba vykonať heapify(úprava zhora dole), zatiaľ čo napríklad pri inserte treba vykonať úpravu zdola hore.

Listing 1: Node v heape

```
1  struct heapNode
2  {
3      int id;
4      int cost;
5  };
```

Listing 2: Odstranenie rootu z heapu

```
1  void removeRoot(struct heapNode **minHeap, int pocetPrvkov)
2  {
3      if(pocetPrvkov != 1) {
4          //Vymenime prvy a posledny
5          int swapCost;
6          int swapId;
7
8          minHeap[1]->cost = minHeap[pocetPrvkov]->cost;
9          minHeap[1]->id = minHeap[pocetPrvkov]->id;
10
11         minHeap[pocetPrvkov] = NULL;
12     }
13     else{
14         minHeap[1] = NULL;
15     }
16 }
```

Listing 3: Uprava heapu po odstraneni rootu

```
1  void heapify(struct heapNode **minHeap, int pocetPrvkov, int pozicia)
2  {
3      int smol = pozicia;
4      int lavy = 2*pozicia ;
5      int pravy = 2*pozicia + 1;
6
7      if (lavy <= pocetPrvkov && minHeap[lavy] -> cost < minHeap[smol] -> ↩
           cost) {
8          smol = lavy;
9      }
10
11     if (pravy <= pocetPrvkov && minHeap[pravy] -> cost < minHeap[smol] -> ↩
           cost) {
12         smol = pravy;
13     }
14
15     if (minHeap[smol] -> cost != minHeap[pozicia] -> cost)
16     {
17         int swapCost = minHeap[smol] -> cost;
18         int swapId = minHeap[smol] -> id;
19
20         minHeap[smol] -> cost = minHeap[pozicia] -> cost;
21         minHeap[smol] -> id = minHeap[pozicia] -> id;
22
23         minHeap[pozicia] -> cost = swapCost;
24         minHeap[pozicia] -> id = swapId;
25
26         heapify(minHeap, pocetPrvkov, smol);
27     }
28 }
```

Djikstrov algoritmus

Djikstrov algoritmus je algoritmus používaný na hladanie najkratšej cesty v ohodnotenom grafe. Algoritmus nám zistí vzdialenosť všetkých vrcholov od vrcholu počiatočného, najčastejšie však určí vzdialenosť od počiatočného k cieľovému vrcholu. Vrcholy prehladávame tak, že navštevujeme ešte neprehľadané vrcholy s najmenšou vzdialenosťou. Na tomto prehľadanom vrchole potom finalizujeme vzdialenosť, zapíšeme odkial bol navštívený a označíme ho za navštívený.

Listing 4: Zaznam v tabulke

```
struct tabulkaPrejdenia
{
    int found;
    int cost;
    int visitedBy;
};
```

Listing 5: Djikstrov algoritmus(pseudokód)

```
while(pocetPrvkov != 0)
    {
        founder = minHeap[1] -> id;
        removeRoot(minHeap, pocetPrvkov);
        if(pocetPrvkov != 0) {
            heapify(minHeap, pocetPrvkov, 1);
        }
        helper = array[founder] -> head;
        while(helper != NULL)
        {
        //Ak este nebol nikdy navstiveny tak ho pridame
        if(tabulkaDjikstra[helper -> id] -> visitedBy == -1 && ↩
           tabulkaDjikstra[helper -> id] -> found != 1){}
         //Ak je cesta kratsia
        else if(tabulkaDjikstra[helper->id]->found != 1 && tabulkaDjikstra↩
           [helper->id]->cost > (helper -> cost + tabulkaDjikstra[founder↩
           ] -> cost ) ){}
         helper = helper -> next;
        }
    }
```

Riešenie zadania

Zadanie sme riešili pomocou Djikstrovho algoritmu s použitím binárnej haldy. Po prevzatí mapy si ju prejdeme a zisťujeme, koľko validných políčok sa v nej nachádza. Ako validné políčka berieme všetky okrem políčka N. Pri tomto procese tiež zistujeme či sa na mape nachádza drak a princezné, a ich počet. Tiež tu kontrolujeme prístup k drakovy a princeznám, musia okolo seba mať aspoň 1 validné políčko.

Na reprezentáciu grafu používame zoznam susednosti reprezentovaný poľom štruktúr, ktoré v sebe obsahujú informácie o vrchole ktorý reprezentujú, a linked list susedných vrcholov.

Po vytvorení grafu susednosti si zavoláme funkciu na djiksrov algoritmus. Funkcia môže vrátiť dĺžku cesty a zapísať cestu do poľa ciest, alebo len vrátť dĺžku cesty. Pri prvom volaní

pri kotrom hľadáme draka cestu rovno zapíšeme do poľa. Po najdení draka si musíme vytvoriť permutácie dostupných princezien, a zistiť ktorá s permutácií je najkratšia. Toto dosiahneme pomocou zavolanie djikstrovho algoritmu na tieto permutácie, ale bez zápisu do našej dráhy. Po zistení najkratšej cesty djikstrov algoritmus spustíme s parametrom pridania do cesty. Po tomto kroku náš program končí.

Možné zlepšenia

- Zmenšiť počet volaní Djikstrovho Algoritmu na permutácie, stačí ho volať na unikátnu princeznú len raz.
- Skončiť djikstrov algoritmus po tom, ako finalizujeme cieľový vrchol.
- Zlepšiť distribúciu princezien v random generácii.

Testovanie

Otestované hraničné prípady :

- case 4: Mapa bez princezien
- Skončiť djikstrov algoritmus po tom, ako finalizujeme cieľový vrchol.
- case 5: Mapa s obklucenim drakom
- case 6: Mapa s obklucenou princeznou
- case 7: Mapa s nepristupnym drakom
- case 8: Mapa s nepristupnou princeznou
- case 9, 10, 11, 12, 13 : 1, 2, 3, 4, 5 princezien
- case 14: 100x100 mapa
- case 15, 16, 17, 18: Rôzne mapy, otestované porovnávaním so spolužiakmi
- case 19: Random generácia
- case 20: Mapa s drakom na začiatku
- case 21: Mapa s princeznou na začiatku
- case 22: Test cien drahy

  Zadanie úspešne splnilo všetky testovacie prípady

Časová zložitosť

Permutácie()

O(n*n!*n) - n = počet prvkov

removeRoot()

O(1)

heapify(), afterInsert(), afterModify()

O(logn) - pri tejto operácii nikdy neprejdeme všetky prvky

findPath() (Djikstra)

O(n + n*logn + 2n) - 1. n je binárna halda

- 2. n*4n je prejdenie vrcholov a ich susedov

- 3. 2n je zápis do poľa + backtracking pri hľadnaní cesty