Adam Žúrek, ID: 79829

Dátové štruktúry a algoritmy

(Zadanie č. 3)

Obsah zadania: Vašou úlohou je implementovať nasledovnú funkciu:

```
int *zachran_princezne(char **mapa, int n, int m, int t, int *dlzka_cesty);
```

Táto funkcia má nájsť cestu, ktorou postupovať, aby popolvár čo najskôr zachránil všetky tri princezné a zneškodnil draka. Mapa je vysoká n políčok a široká m políčok ($1 \leq n,m \leq 1000$). Parameter t ($0 \leq t \leq 106$) určuje, kedy od začiatku vášho hľadania sa drak zobudí a zje prvú princeznú. Keďže drak lieta veľmi rýchlo, spoľahnite sa, že ak sa vám ho nepodarí zneškodniť dovtedy ako sa zobudí, princezné už nezachránite. Prechod cez lesný chodník trvá jednu jednotku času a drak sa zobudí v t-tej jednotke času, kedy už bude neskoro. A nezabudnite, že najprv musíte zneškodniť draka, až potom môžete zachraňovať princezné (hoci by ste aj prechádzali predtým cez políčko kde je princezná). Veď ako by to bolo, keby ste bojovali s drakom pri zástupe princezien ako diváčkami...

Teleporty začnú fungovať až potom ako zapnete generátor energie pre teleporty. Každý teleport má identifikátor (jednu cifru: od 0 po 9) a nejakým teleportom sa môžete preniesť na ľubovoľné iné políčko s rovnakým identifikátorom teleportu. Keď teda nastúpite na teleport 0, tak sa môžete preniesť na akýkoľvek iné políčko s teleportom s identifikátorom 0. Prenos je okamžitý (trvá nula časových jednotiek).

Nájdenú cestu vráťte ako postupnosť súradníc (dvojíc celých čísel x,y, kde $0 \leq x < n$ a $0 \leq y < m$) všetkých navštívených políčok. Na začiatku sa vždy nachádzate na políčku so súradnicami 0,0 a na poslednom navštívenom políčku musí byť jedna z troch princezien. Ak existuje viacero rovnako dlho trvajúcich ciest, môžete vrátiť ľubovoľnú z nich. Výstupný argument dlzka_cesty nastavte na počet súradníc, ktoré ste vrátili ako návratovú hodnotu.

Implementujte vyššie uvedenú funkcionalitu čo možno najefektívnejšie.

Adam Žúrek, ID: 79829

Vypracovanie: Základ môjho riešenia je postavený na Dijkstrovom algoritme, ktorý hľadá najkratšiu cestu v hranovo ohodnotenom grafe. Pri prechádzaní a vyhľadávaní cesty musí brať navyše do úvahy prekážky, ako nepriechodná prekážka (N) a hustý lesný porast (H). Popolvár začína na súradniciach [0,0] a musí čím najskôr zabiť draka. Ak sa na mape nachádza generátor, program porovnáva, či sa oplatí pomocou Dijkstrovho algoritmu ísť najskôr ku generátoru. Ak áno, popolvár pôjde cez teleport k drakovi, inak ho zapne vtedy, keď draka zabije. Následne zachraňuje princezné jednu po druhej.

Použité dátové štruktúry v mojom riešení sú graf (graph) a minimálna halda (heap).

Rozbor jednotlivých funkcií:

```
void heapInsert(VERTEX *v)
```

slúži na pridanie prvka do haldy. Funkcia heapInsert pridá prvok na koniec haldy, inkrementuje globálnu premennú ukazujúcu na koniec a prebuble pridaný prvok kým nenarazí na parenta, ktorý je menší, ako pridaný prvok.

```
VERTEX *heapDelete()
void heapify(int i)
```

za pomoci týchto funkcií program zmaže prvý (najmenší) prvok v halde. Funkcia heapDelete vymení prvý prvok za posledný a následne globálnu premennú ukazujúcu na koniec haldy dekrementuje. Zavolá funkciu heapify, ktorá rekurzívne prebuble prvý prvok na správnu pozíciu, dokiaľ obe deti nebudú väčšie, alebo existuje iba jedno väčšie, alebo ak neexistuje žiadne.

```
int dijkstra(int srcX, int srcY, int destX, int destY, char **map_d, int
map_h, int map_w)
```

funkcia hľadajúca najkratšiu cestu medzi súradnicami začiatku (source) a konca (destination) grafu, zostaveného z mapy v parametri funkcie, jej výšky a šírky. Prehľadávanie začína v súradniciach začiatka (srcX, srcY) a zakaždým, kým sa pridá ďalší prvok do haldy, sa kontroluje pomocou funkcie

```
void direction(VERTEX *v, char **map, int map_h, int map_w)
```

či, susedné súradnice niesú N alebo H. Takto postupuje program dokiaľ sa nedostane k súradniciam konca a teda (destX, destY).

```
void printfpath(int srcX, int srcY, int destX, int destY)
```

funkcia pridávajúca do poľa x-ové a y-ové súradnice, po ktorých sa popolvár rozhodol ísť zo začiadku (source) do konca (destination). Nakoniec, funkcia zachran_princezne() vráti toto pole, ako kompletnú cestu jeho dobrodružstva.

```
int *zachran_princezne(char **mapa, int n, int m, int t, int *dlzka_cesty)
```

hlavná funkcia, ktorá spúšta ostatné funkcie. Tu sa z mapy získajú súradnice všetkých dôležitých bodov na mape (drak, princezné, generátor). Následne sa kontroluje, po akej ceste je efektívnejšie ísť podľa volaní Dijkstrovho algoritmu.

Záver: Moje riešenie zbehlo na všetkých scenároch testovača, kde scenáre s číslom 3,4 a 8 zbehli v najlepšom možnom čase. Výhodou môjho riešenia je pomerne jednoduchá implementácia, čitateľnosť zdrojového kódu a časová náročnosť. Za výraznú nevýhodu, ktorá časovú náročnosť aj výrazne ovplivnila považujem deficit v logike porovnávania dĺžky ciest medzi princeznami. Správna implementácia by mala za následok splnenie všetkých scenárov v najlepšom čase. Bohužial z časového hľadiska sa mi nepodarilo efektívne doimplementovať (aby tester zbehol v čase) túto opravu.

Obrázky scenárov, ktoré som testoval ešte pred spustením testovača aj s výstupom môjho programu dodám v prílohe.

Časová a pamäťová zložitosť:

```
void heapInsert(VERTEX *v)
```

- Časová zložitosť: O(log N)
- Pamäťová zložitosť: P(1)

```
VERTEX *heapDelete()
void heapify(int i)
```

- Časová zložitosť: O(log N)
- Pamäťová zložitosť: P(1)

```
int dijkstra(int srcX, int srcY, int destX, int destY, char **map_d, int map_h, int map_w)
```

- Časová zložitosť: $O(n^2)$
- Pamäťová zložitosť: P(n)

```
void direction(VERTEX *v, char **map, int map_h, int map_w)
```

- Časová zložitosť: O(1) ak je generátor vypnutý, alebo neexistuje, inak $O(n^2)$
- Pamäťová zložitosť: P(1)

```
void printfpath(int srcX, int srcY, int destX, int destY)
```

- Časová zložitosť: $O(n^2)$
- Pamäťová zložitosť: P(n)

```
int *zachran_princezne(char **mapa, int n, int m, int t, int *dlzka_cesty)
```

- Časová zložitosť: O(1)
- Pamäťová zložitosť: P(1)