# **Popolvár**

Autor: Jozef Varga

# **Hlavná myšlienka**

Zadanie je vypracované v jazyku C. V riešení využijeme pri hľadaní najkratšej cesty dijkstrov algoritmus a minimálnu haldu v ktorej budem ukladať políčka z mapy (x a y súradnice) ktoré dostaneme od dijkstra. Výslednú cestu zostavíme pomocou čiastočných ciest ktoré nám vytvorí dijkstra. Dijkstra prejde celú mapu a nájde nám zo všetkých bodov najlacnejšiu cestu do daného bodu. Cesty ktoré sú pre nás podstatne (Popolvár <–> Drak, Drak <–> Princezna, a.t.ď.) si uložíme. Dijkstra spustíme viac krát aby sme dostali všetky potrebné cesty. Cesty ktoré sme si uložili spracujeme (pomocou permutácií) a zistíme ktorá z kombinácií (či ísť najskôr ku princeznej č.1 alebo č.2) je najvhodnejšia (najlacnejšia cesta). Po vyhodnotení, ktorá z ciest je najvhodnejšia danú cestu zostavíme.

# Riešenie

Použité knižnice:  
#include<stdio.h>  
#include<stdlib.h>  
  
Vlastné štruktúry:  
typedef struct vrchol{  
    int vzdialenost;  
    int x;  
    int y;  
    int pred;  
}VRCHOL;  
  
Táto štruktúra nám v programe predstavuje jedno políčko mapy a jej obsah je nasledovný:  
vzdialenost – hovorí o cene cesty ku bodu od ktorého sme pustili dijkstru  
x – je to x-ová súradnica v poli  
y – je to y-ová súradnica v poli  
pred – ukazuje nám na predchádzajúce políčko v danej ceste  
  
Globálne premenné:  
int \*\*teleport0, \*\*teleport1, \*\*teleport2, \*\*teleport3, \*\*teleport4, \*\*teleport5, \*\*teleport6, \*\*teleport7, \*\*teleport8, \*\*teleport9;  
  
V programe využívame 10 globálnych premenných a to teleport0-teleport9. Tieto premenné sú 2 rozmerné polia. Každá premenná uchováva informácie o políčkach, na ktorom je teleport (x a y súradnicu) a jednu informáciu, či už daný druh teleportu (teleport 0-9) bol použitý.

# Funkcie

void vloz\_halda(VRCHOL \*\*halda, VRCHOL \*novy, int \*pocet\_v\_halde){}

* táto funkcia vklada VRCHOL novy (políčko) do minimálnej haldy. Halda je interpretovaná jednorozmerným poľom. Prvok dáme na koniec haldy a následne ho prebublávame, teda ak je prvok menší ako vrchol ktorý je nad nim, tak ich vymeníme. Toto opakujeme pokiaľ prvok nie je na správnej pozícií

VRCHOL\* vyber\_halda\_min(VRCHOL \*\*halda, int \*pocet\_v\_halde){}

* táto funkcia vracia VRCHOL (políčko) s najmenšou vzdialenosťou. Ten sa v halde nachadza na vrchu. Po vybratí prebublávame po deťoch nadol podľa vzdialenosti. Ak je ľavé dieťa menšie ako pravé dieťa, tak ľavé dieťa dáme na daný vrchol a ideme na jeho pozíciu. Tak to opakujeme pokial sa nedostaneme na koniec stromu.

int cena\_policka(char znak){}

* zisťuje cenu políčka na mape (dĺžka presunu cez políčko). Ak je na políčku ‚H‘ alebo ‚h‘ cena je 2, ak je ‚N‘ ‘n‘ cena je -1 (nedosiahnuteľné) a pri všetkých ostatných prípadoch funkcia vráti cenu 1

int \*\*vrat\_teleport(char znak, int \*x){}

* vráti mi teleport podľa políčka na akom stojím (ak je na políčku ‚0‘ vráti teleport0)

void nastav\_teleport(char znak, int \*\*teleport){}

* nastaví globálnu premennú teleport podľa znaku, aký príde. Teda ak pošlem znak ‚0‘ tak globálnu premennú teleport0 nasmeruje na nami poslanú premennú teleport

void presun\_hodnot\_teleportu(int \*\*t0,int \*\*t1,int \*\*t2,int \*\*t3,int \*\*t4,int \*\*t5,int \*\*t6,int \*\*t7,int \*\*t8,int \*\*t9){}

* nastaví všetky globálne parametre (využíva sa ako záloha pri práci s globálnymi premennými)

void dijkstra(char \*\*mapa, VRCHOL \*\*pole\_ciest, int x, int y, int n, int m, int teleportovanie){}

* táto funkcia nám vynuluje a nastaví pole\_ciest (jednorozmerná mapa) ktorá obsahuje štruktúry VRCHOL. Nastavuje každému VRCHOLU z pole\_ciest x a y súradnicu, taktiež vzdialenosť na -2 a pred na -1 aby sme vedeli či dané políčko bolo skontrolované. Následne kontroluje od súradníc ktoré sme mu poslali ako parameter všetky políčka vo všetkých smeroch. Napríklad z políčka x y sa pozrie doprava a ak môžem, nastavím ho a vložím to políčko do haldy (kontrolu nastavenie a vloženie do haldy vykonáva funkcia nastavenie\_vedlajsieho\_policka()).Potom pozriem doľava taktiež ak môžeme nastaviť a vložiť dané políčko do haldy vykonáme to. To isté spravíme aj vertikálne. Na konci skontrolujeme, či halda je prázdna, ak nie je tak vyberiem z nej prvú hodnotu (políčko s najmenšou vzdialenosťou) a opakujem kontrolovanie (horizontálne aj vertikálne) pokiaľ nebude halda prázdna. Tým, že vkladám len políčka, ktoré sú nenavštívené, tak po prejdení celej mapy už nebude čo vkladať do haldy a budeme z nej len vyberať. Tým pádom haldu celkom vyprázdnime a ukončíme cyklus. Funkcia nevracia nič, no pole\_ciest ostane naplnené (prvky predstavujú mapu a obsahuju všetky potrebné informácie na zostavenie cesty). Táto funkcia má parameter teleportovanie, ktorý hovorí, či dijkstra využíva aj teleporty (1) alebo nie (0). Ak ich využíva, jedinou zmenou je to, že pred kontrolou do haldy zavolá funkciu teleport\_dijkstra(), ktorá skontroluje, či políčko na ktorom stojíme je teleport (bližšie informácie nižšie).

void teleport\_dijkstra(char \*\*mapa,VRCHOL \*\*pole\_ciest, VRCHOL \*\*halda, VRCHOL \*vyber, int n, int m, int \*pocet\_v\_halde){}

* ak pri vykonávaní dijkstrovho algoritmu narazíme na políčko s teleportom, spustíme túto funkciu, ktorá zisťuje či už teleportyX (napr. ak sme na políčku ‚0‘ tak teleport 0) už bol využitý, ak nie tak nahádže všetky ešte neskontrolované polia tohto teleportu do haldy a nastaví ich ako pri funkcii nastavenie\_vedlajsieho\_policka.

void nastavenie\_vedlajsieho\_policka(char \*\*mapa, VRCHOL \*\*pole\_ciest, VRCHOL \*\*halda,int zx, int zy, VRCHOL \*vyber, int n, int m, int \*pocet\_v\_halde){}

* túto funkciu volá dijkstrov algoritmus pri presúvaní sa po políčkach mapy. Funkcia zisťuje, či sme dané políčko už skontrolovali a dali do haldy, ak nie tak ho nastavíme a následne vložíme do haldy. Ak je dané políčko ‚n‘ alebo ‚N‘ tak ho vynecháme. Tieto políčka sú nepriechodné. Pred vkladaním nastavujeme x,y súradnicu, vzdialenosť čo je vlastne vzdialenosť od políčka z ktorého tu ideme plus cena môjho políčka. Ako posledné nastavíme predchodcu, čo je vlastne čislo políčka (v poli) z ktorého tu ideme.

int \*\*vyhladaj\_cestu(VRCHOL \*\*pole\_ciest, int ciel\_x, int ciel\_y, int start\_x, int start\_y, int n, int m){}

* funkcia nám vracia dvojrozmerné pole obsahujúce x a y súradnice cesty. Funkcia vie súradnice prvku od ktorého má cestu vytvárať. Každý prvok v pole\_ciest má predchodcu (číslo ukazujúce na predchádzajúci VRCHOL v danom poli) pomocou ktorého sa vieme dostať na začiatok (súradnicu od ktorej sme pustili dijkstru). Touto cestou postupne prechádzame a ukladáme do výsledného dvojrozmerného pola x a y súradnice

int \*stavba\_cesty(int \*cesta, int \*\*cast, int \*x){}

* funkcia vkladá do poľa cesta za sebou x a y súradnicu z dvojrozmerného poľa časť (x-ová súradnica -> cast[0][0] = 1, y-ová súradnica -> cast[1][0] = 0, výsledné pole -> cesta[0] = 1, cesta[1] = 0). Do poľa cesta sa to ukladá pomocou parametra x, ktorý hovorí, kde v jednorozmernom poli začíname ukladať ( túto funkciu môžeme napríklad tri krát za sebou spustiť a nastaviť aby tú istú cestu zapísalo za sebou do poľa cesta, teda nezapisujeme od začiatku poľa ale zapisujeme od premennej x).

int \*stavba\_cesty\_opacne(int \*cesta, int \*\*cast, int \*x){}

* funkcia robí to isté ako funkcia stavka\_cesty() len zapisuje v opačnom poradí. Teda ak máme cestu od Draka ku Princeznej, no potrebujeme opačnú cestu, tak využijeme túto funkciu.

int \*vytvor\_cestu(int velkost, int \*\*prve,int a, int \*\*druhe,int b,int \*\*tretie, int c,int \*\*stvrte, int d,int \*\*piate, int e, int \*x){}

* funkcia podľa parametrov vytvára celkovú cestu. Parametre určujú, či využívame pri stavbe cesty funkciu stavba\_cesty\_opacne() alebo stavba\_cesty(). Vieme že cesta sa môže skladať s maximalne 5 častí preto máme ako parametre 5 ciest, 5 prepínačov ktoré určujú ako zapísať cestu (stavba\_cesty() - 1,stavba\_cesty\_opacne() - 0). Posledný prepínač hovorí o celkovom počte súradníc výslednej cesty.

void kontrola\_teleport(char \*\*mapa, int i, int j){}

* funkcia slúži na pridanie súradníc teleportu do globálnej premennej

void nastavenie\_bodov(int \*bod,int x,int y){}

* funkcia nastavuje bod a jeho x, y súradnicu

int \*zachran\_princezne(char \*\*mapa, int n, int m, int t, int \*dlzka\_cesty){}

* táto funkcia prehľadá celú zadanú mapu a ukladá súradnice 3 princezien, draka, popolvára, generátora a všetkých teleportov. Ak nájdeme všetky potrebné súradnice, spustíme funkciu dijkstra() a následne vytvoríme všetky potrebné cesty od daného bodu (vyhladaj\_cestu()) z ktorého púšťame dijkstru. Dijkstru púšťame s využitím teleportov alebo bez využitia teleportov, zo súradnicami generátora, draka a 2 princezien, čiže dokopy 8 krát. Po vytvorení všetkých čiastkových ciest zistíme najkratšiu kombináciu ako zabiť draka a zachrániť princezné (s využitím, alebo bez využitia teleportov).  Ako posledné celú cestu pospájame pomocou funkcie vytvor\_cestu() a vysledné súradnice cesty pošleme s funkcie von.

# Testovanie

Testovanie prebiehalo po častiach ako som tvoril program. Ako prvé som vytvoril minimálnu binárnu hladu, ktorú som testoval tak, že som do nej vkladal a vyberal hodnoty pokiaľ som si nebol istý že halda funguje správne. Po vytvorení haldy som skúšal funkciu zachran\_princezne(), ktorá mala zatiaľ iba jednu úlohu a to nájsť potrebné objekty na mape (generátor, princezné, draka). Na začiatku som sa teleportom veľmi nevenoval, preto som ich ani nevyhľadával. Následne som vytvoril prehľadávanie (dijkstra) mapy a vyskúšal som ho na vlastných mapách, napríklad:  
  
int n = 5;  
int m = 5;  
char \*\*mapa;  
mapa = (char\*\*)malloc(n \* sizeof(char\*));  
mapa[0] = "HHCNP";  
mapa[1] = "CCCPN";  
mapa[2] = "GHHHN";  
mapa[3] = "NNPHC";  
mapa[4] = "PCCDH";  
  
Keď som si bol istý, že prehľadávanie funguje správne (zisťoval som to výpismi predchádzajúcich bodov), pokračoval som vytvorením cesty. Cestu som taktiež kontroloval na týchto mapách, pokiaľ som si nebol istý jej správnosťou. Keď bolo všetko správne, skúsil som doimplementovať teleporty, ktoré som taktiež skúšal na mapách, ale aj s využitým teleportov napr:  
  
int n = 5;  
int m = 5;  
char \*\*mapa;  
mapa = (char\*\*)malloc(n \* sizeof(char\*));  
mapa[0] = "H0CNP";  
mapa[1] = "GCCP1";  
mapa[2] = "CHHHN";  
mapa[3] = "NNPHC";  
mapa[4] = "P0CD1";  
  
Toto testovanie ma ubezpečilo v správnosti môjho programu.

# Zložitosti

Časová zložitosť:

* Náš program využíva binárnu haldu, ktorá má 2 funkcie a to vloženie do haldy a následný výber. Výber z haldy má zložitosť O(1), no jej následné preusporiadanie má zložitosť O(log(n)) (je to vlastne binárny strom). Vklad do haldy má zložitosť O(log(n)). Čiže pri výbere a vkladaní všetkých prvkov je to O((x.y).log(x.y)). Ďalej v našom programe využívame Dijkstrov algoritmus, ktorý prechádza všetky prvky a  vkladá/vyberá ich do/z haldy. Tým je jeho zložitosť O(n.log(x.y)). Celková zložitosť funkcie zachran\_princezné() je O(n.(x.y).log(x.y)) pričom n je počet spustených dijkstrových algoritmov, čo je konštanta, takže celková zložitosť funkcie je O((x.y).log(x.y)).

Priestorová zložitosť:

* Priestorová zložitosť našej funkcie je O(x.y)

x-počet riadkov mapy  
y-počet stĺpcov mapy

# Zhodnotenie

Tento program ma veľa naučil, čo sa týka vyhľadávacích algoritmov a ich využitia, čo si osobne myslím, že je veľmi podstatná záležitosť. Hlavne som sa zameral na dijkstrov algoritmus a jeho implementáciu. Moje riešenie nájde najlacnejšiu cestu avšak iba pre náš prípad problému, teda že máme práve 3 princezné, 1 popolvára, 1 draka a 1 generátor. Ak by sme týchto „objektov“ mali viac, náš algoritmus by sme museli postaviť ináč . Napríklad vytvorením dijkstra, ktorý sám spúšťa dijkstrov s rozdielnymi označeniami. Avšak pre problém, ktorý nám bol zadaný, tento algoritmu bohate vyhovuje. Myslím si, že moje riešenie je správne, efektívne a rýchle. Tento príklad mi ukázal nové možnosti vyhľadávania a následného spracovania vyhľadaných dát.