

Tomáš Rafaj

Popolvár - Zadanie 3

Dátové štruktúry a algoritmy

Zámer a cieľ zadania 3

Cieľom zadania 3 bolo naprogramovať zadanú funkciu :

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

Jej funkčnosť spočívala v tom, že ako argument prišla mapa rozmeru $n \times m$, kde $0 < n, m < 101$. Cieľom bolo najprv nájsť najkratšiu cestu ku drakovi za čas, ktorý to buď popolvár stihol alebo nie. V prípade, že to nestihol v dostupnom čase tak testovací main, ktorý bol priložený v dokumentácii to vypíše a je potrebné zvýšiť dostupný čas lebo Popolvár to urobil za najmenší možný čas, ktorý sa dal. V prípade, že všetko bolo v poriadku tak sa vybral hľadať od draka všetky princezné. A cieľom je za najkratšiu cestu nájsť všetky princezné (na poradie sa medze nekladú), pričom popolvár skončí svoju cestu na pozícii jednej z nich. A funkcia má vrátiť pointer na intové pole, kde sú uložené jednotlivé indexy cesty, ktorú absolvoval Popolvár. Princezien je najviac 5, najmenej 1. Drak je iba jeden.

Vlastná implementácia a realizácia zámeru zadania

Na začiatku zadanej funkcie zachran_princezne som si urobil viaceré dôležité inicializácie. Z jednou prvých inicializácií bolo rozhodne zadefinovanie permutácií potrebných k vykonaniu všetkých zoradení princezien. Urobil som teda permutácie perm5 pre prípad 5 princezien, perm4 pre prípad 4, perm3 pre 3, perm2 pre 2 ale permutácia 1 je 1, čiže to nebolo potrebné tvoriť permutácie.

```
int perm5[120][5] = {
    {1,2,3,4,5}, {2,1,3,4,5}, {3,1,2,4,5}, {1,3,2,4,5}, {2,3,1,4,5},
    {3,2,1,4,5}, {4,2,3,1,5}, {2,4,3,1,5}, {3,4,2,1,5}, {4,3,2,1,5},
    {2,3,4,1,5}, {3,2,4,1,5}, {4,1,3,2,5}, {1,4,3,2,5}, {3,4,1,2,5},
    {4,3,1,2,5}, {1,3,4,2,5}, {3,1,4,2,5}, {4,1,2,3,5}, {1,4,2,3,5},
    {2,4,1,3,5}, {4,2,1,3,5}, {1,2,4,3,5}, {2,1,4,3,5}, {5,1,2,3,4},
    {1,5,2,3,4}, {2,5,1,3,4}, {5,2,1,3,4}, {1,2,5,3,4}, {2,1,5,3,4},
    {3,1,2,5,4}, {1,3,2,5,4}, {2,3,1,5,4}, {3,2,1,5,4}, {1,2,3,5,4},
    {2,1,3,5,4}, {3,5,2,1,4}, {5,3,2,1,4}, {2,3,5,1,4}, {3,2,5,1,4},
    {5,2,3,1,4}, {2,5,3,1,4}, {3,5,1,2,4}, {5,3,1,2,4}, {1,3,5,2,4},
    {3,1,5,2,4}, {5,1,3,2,4}, {1,5,3,2,4}, {4,5,1,2,3}, {5,4,1,2,3},
    {1,4,5,2,3}, {4,1,5,2,3}, {5,1,4,2,3}, {1,5,4,2,3}, {2,5,1,4,3},
    {5,2,1,4,3}, {1,2,5,4,3}, {2,1,5,4,3}, {5,1,2,4,3}, {1,5,2,4,3},
    {2,4,1,5,3}, {4,2,1,5,3}, {1,2,4,5,3}, {2,1,4,5,3}, {4,1,2,5,3},
    {1,4,2,5,3}, {2,4,5,1,3}, {4,2,5,1,3}, {5,2,4,1,3}, {2,5,4,1,3},
    {4,5,2,1,3}, {5,4,2,1,3}, {3,4,5,1,2}, {4,3,5,1,2}, {5,3,4,1,2},
    {3,5,4,1,2}, {4,5,3,1,2}, {5,4,3,1,2}, {1,4,5,3,2}, {4,1,5,3,2},
    {5,1,4,3,2}, {1,5,4,3,2}, {4,5,1,3,2}, {5,4,1,3,2}, {1,3,5,4,2},
    {3,1,5,4,2}, {5,1,3,4,2}, {1,5,3,4,2}, {3,5,1,4,2}, {5,3,1,4,2},
    {1,3,4,5,2}, {3,1,4,5,2}, {4,1,3,5,2}, {1,4,3,5,2}, {3,4,1,5,2},
    {4,3,1,5,2}, {2,3,4,5,1}, {3,2,4,5,1}, {4,2,3,5,1}, {2,4,3,5,1},
    {3,4,2,5,1}, {4,3,2,5,1}, {5,3,4,2,1}, {3,5,4,2,1}, {4,5,3,2,1},
    {5,4,3,2,1}, {3,4,5,2,1}, {4,3,5,2,1}, {5,2,4,3,1}, {2,5,4,3,1},
    {4,5,2,3,1}, {5,4,2,3,1}, {2,4,5,3,1}, {4,2,5,3,1}, {5,2,3,4,1},
    {2,5,3,4,1}, {3,5,2,4,1}, {5,3,2,4,1}, {2,3,5,4,1}, {3,2,5,4,1}
};
```

Následne došlo k inicializácii prakticky všetkých potrebných pomocných a iných premenných. Premenné ako i,j,k,x,f slúži na úchovu indexov premenných. Pom1 a pom2 slúžili pri zisťovaní pozícií v matici. Vytvoril som viacero 2d polí, ktoré som rôzne využíval. Začnem opisom pola mapaExtra a budem pokračovať až po koniec zadanej funkcie, kde už je nájdená finálna najkratšia cesta.

```
int i;
int j;
int x;
int k;
int f;
int drak;
int helper;
int pom1, pom2;
int **mapaExtra;
int **mapaMATRIX;
int **mapaMATRIXfinal;
int *poleHciek;
int *cestovnaMapa;
int indexCMapy = 0;

int **prin;
prin = (int**)malloc(5*sizeof(int*));
for(i=0; i<5; i++)
    prin[i] = (int*)malloc(2*sizeof(int));

mapaExtra = (int**)malloc(n*sizeof(int*));
for(i=0; i<n; i++){
    mapaExtra[i] = (int*)malloc(m*sizeof(int));
}
```

Array mapaExtra

2D Array s názvom mapaExtra slúžilo k tomu, že ku pôvodnej zadanej forme mapy, ktorá prišla argument vytvorilo vhodnejšiu formu tejto mapy tým, že všetky C,H,D,P,N rozlíšilo a prepísalo do vhodnejšej formy na int, pretože práca s tým ako s array-om charov by bola rozhodne menej príjemná. Preto si dovoľím uviesť vzorový príklad ako vlastne prebiehala konverzia z pôvodnej mapy na tú novú, jednoduchšie prístupnú. Naľavo je pôvodná mapa, napravo opravená.

Urobil som teda jednoduchý prechod cez mapu pomocou for cyklu, kde som vlastne prešiel celú pôvodnú zadanú mapu a následne pri náleze som vykonal potrebnú konverziu.

```
for(i=0;i<n;i++)
{
    for(j=0;j<m;j++)
    {
        if(mapa[i][j]=='C')
        {
            mapaExtra[i][j]=c;
            c++;
        }
    }
}
```

V prípade, že pri prechode mapy je niektorý štvorček rovný charu C, tak sa do novej mapy na toto miesto zapísala hodnota premennej c, ktorá mala pôvodnú hodnotu 200000, H malo 100000, D 50000, P malo 60000 tisíc a postupne sa tie hodnoty inkrementovali. V prípade, že som teda narazil na prvé c, zapísal som na jeho pozíciu hodnotu číslo 200000, zvýšil som ho na 200001 aby pri ďalšom zbehnutí zapísalo iné číslo, to nebolo nutné ale pomôže mi to lepšie orientovať sa v mape. Lebo na základe modulovania zistím koľké C v poradí to je v pôvodnej mape a kde sa nachádza. Jediná zmena, kde sa inkrementácia nerobila bolo N, kde som všetko označoval -1, teda menšie ako 0 a tieto body som neskôr ani nebral do úvahy ako možnú cestu. Zbytočné tvoriť maticu, resp. graf s hranami, do ktorých sa aj tak nepôjde a nikam nevedú. Po vykonaní tohto úkonu sme už teda vedeli rozlíšiť na základe čísla na danej pozícii, o ktoré písmeno sa jedná a koľké v poradí je. 200000 a 100000 som zvolil z toho dôvodu, že mapa môže byť max 100x100 podľa zadania, čo je 10000 tisíc, a preto som si dal rezervu radšej 100 tisíc aby sa mi jednotlivé číselné priradenia C a H nekřížili.

C	C	C	H
C	H	H	P
D	N	H	N
C	C	N	N

200000	200001	200002	100000
200003	100001	100002	60000
50000	-1	100003	-1
200004	200005	-1	-1

[illegible][illegible][illegible]

Tomáš Rafaj – Popolvár – Zadanie 3

Následne som ešte urobil to, že teraz už nechcem pracovať s pomocnými riadkami a pomocnými stĺpcami a preto som si urobil ešte jedno 2d pole ale o jedno kratšie a jedno menej širšie a tam som zapísal iba časť bez pomocného riadku a stĺpca. Tento array som nazval mapaMATRIXfinal, s ktorým už budem pracovať počas celého behu programu a budem ho posielať aj do dijkstru. Takto sa bude zasielať array mapaMATRIXfinal do dijkstru.

0	1	0	0	1	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0	0
0	1	0	1	0	0	1	0	0	0	0	0
0	0	2	0	0	0	0	2	0	0	0	0
1	0	0	0	0	1	0	0	1	0	0	0
0	2	0	0	2	0	2	0	0	0	0	0
0	0	2	0	0	2	0	2	0	2	0	0
0	0	0	1	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	0	0	2	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	1
0	0	0	0	0	0	0	0	0	0	1	0

Toto bola akoby prvá časť programu, kde bola základná a potrebná inicializácia vo funkcii zachran_princezne a vytvorila sa taktiež matica, označili sa všetky písmená číslami / ID-čkami.

Funkcia algoritmus

Táto funkcia má viacero vstupných argumentov.

```
int algoritmus(int **matrix,int n,int source, int findThis,int *poleHciek, int pocetH,
,
int *cestovnaMapa, int *indexCMapy, int urobit,int** princezne, int pocetP)
```

Takže argument číslo 1 označený int** matrix je vlastne matica vypočítaná vo funkcii zachran_princezne.

Argument číslo 2 int n označuje počet riadkov a stĺpcov v matici, ktorá prišla ako argument číslo 1.

Argument číslo 3 int source označuje odkiaľ dijkstra začína vyhľadávanie, z ktorého stĺpca / riadka.

Argument číslo 4 int findThis označuje, ktorý stĺpec / riadok v matici hľadáme z argumentu č.3.

Argument číslo 5 int* poleHciek obsahuje array, ktorý obsahuje jednotlivé stĺpce / riadky , teda označenie, kde bude treba ešte urobiť extra operáciu aby to zarátalo väčší čas pre H-čka.

Argument číslo 6 int pocetH reprezentuje dĺžku array-u v argumente 5.

Argument číslo 7 a 8 bude slúžiť na zapísanie cesty.

Argument číslo 9 je buď 1 alebo 0, čo si symbolizuje, či zapisovať alebo nezapisovať cestu
(0 nezapisovať, 1 zapisovať)

Argumenty číslo 10 a 11 sú pozície princezien a ich počet na ošetrenie ich nedostupnosti.

Na začiatku funkcie sa zasa urobia začiatkové inicializácie.

```
int tempIndex = 0;
int z;
int cena = 0;
int count = 1,temp,next,i,j,k;

int *tempArray;
tempArray = (int *)malloc(INFINITY*sizeof(int));

int **cost;
cost = (int**)malloc(n*sizeof(int*));
for(k=0;k<n;k++)
    cost[k] = (int *)malloc(n*sizeof(int));

int **info;
info = (int**)malloc(n*sizeof(int*));
for(k=0;k<n;k++)
    info[k] = (int *)malloc(3*sizeof(int));
```

Prvé 4 riadky sú iba pomocné na uchovávanie rôznych premenných, country, swapovanie a podobné. Prvý array tempArray slúži na uchovávanie cesty, kam si najprv uloží backtracknutú cestu. Array cost bude nositeľom cien stĺpcov a riadkov. Čiže v prípade source 0 je cost[0] určite 0, pretože sám do seba má cenu 0. Ostatné ceny zapíšeme na základe matice. V prípade, že tam nevedia žiadna cesta, zapisujeme INFINITY.

```

for(i=0;i<n;i++){
    for(j=0;j<n;j++){
        if(matrix[i][j]!=0){
            cost[i][j]=matrix[i][j];
        }
        else cost[i][j]=INFINITY;
    }
}

```

Array info je 2d pole, ktoré pre každý riadok a stĺpec vytvára údajový zápis, toto sa dalo urobiť aj cez štruktúru ale mne bolo sympatickejšie to urobiť cez 2d pole. Čiže pre každý vrchol (počet riadkov / stĺpcov) si uchováva 3 údaje. Pristupujeme k tomu takto ak chceme získať údaje od 0tého riadku / stĺpca tak pristupujeme pomocou info[0][0-2] , kde druhá časť predstavuje informácie.

Informácie sú zapisované spôsobom, že na nultom indexe si zanechávame 1 alebo 0, že či bol daný vrchol navštívený, v prípade, že áno dáme mu 1, ak ešte nebol má tam 0, čo na začiatku nastavíme všetkým. V indexe číslo 1 je zapísaná cena od začínajúceho vrcholu / source-u. V indexe číslo 2 je zapísaný predošlý vrchol, toto nie je nutné ale z dôvodu, že chceme následne backtrackovať cestu až k začiatku, bude to potrebné zapisovať. Tuto vidíme inicializáciu, najprv je všetko previous source, ceny sú buď infinity alebo je tam konekcia, visited sú všetky 0.

```

for(i=0;i<n;i++)
{
    info[i][0] = 0;
    info[i][1] = cost[source][i];
    info[i][2] = source;
}

```

Následne idem urobiť pre celé pole dijkstru. To znamená, že najprv nastavíme minimum na INFINITY, a postupne prebehne všetky vrcholy kde pozeráme, či je to menšie ako infinity a či boli unvisited, ak to splňa tieto podmienky, nastavíme next na tento vrchol, a do hodnoty minima (temp-u) nastavíme už novú vzdialenosť, tá ktorá nie je INFINITY, čiže je tam spojenie. Po zbehnutí cyklu, to čo ostane ako najlacnejšie nastavíme na visited.

Následne urobíme ďalší cyklus a pre všetky nenavštívené, ktorých aktuálna cena uložená v tempe + cena nextu v stĺpci dáva menšiu ako už predtým prípadne zapísaná hodnota, na začiatku to je INFINITY. Ak to splní, prepíšeme danú cenu na novú cenu ako súčet tempu a ceny vyhovujúceho nextu, ako predošlý prvok, bude teda next. A týmto v cykle urobíme vlastne kompletného dijkstru a už ostáva len backtracking zistiť cestu.


```

for(j=0;j<n-1;j++)
{
    temp=INFINITY;

    for(i=0;i<n;i++)
    {
        if( info[i][1] < temp && info[i][0]==0 )
        {
            next=i;
            temp = info[i][1];
        }
    }

    info[next][0]=1;

    for(i=0;i<n;i++)
    {
        if( info[i][0] == 0 )
            if( temp + cost[next][i] < info[i][1] )
            {
                info[i][1] = temp + cost[next][i];
                info[i][2] = next;
            }
    }
}

```

Následne som urobil backtracking k zisteniu, úplnej ceny cesty, čo mi pomohlo odhaliť najkratšiu, resp. najlacnejšiu cestu. Cenu som rátať pomocou inkrementácie pomocnej s názvom cena.

```

for(i=0;i<n;i++)
{
    if(i==findThis && i != source)
    {
        if(urobit==1)tempArray[tempIndex] = i;
        if(urobit==1)tempIndex++;
        j=i;
        while(1)
        {
            if(j==source)
            {
                break;
            }
            for(z=0;z<pocetH;z++){
                if(j==poleHciek[z]){
                    cena+=2;
                    z=0;
                    break;
                }
                else continue;
            }

            if(z==pocetH)cena++;
            j = info[j][2];
            if(urobit==1)tempArray[tempIndex] = j;
            if(urobit==1)tempIndex++;
        }
    }
}

```

V prípade potreby zapísania cesty sa poslala jednotka ako argument, kde sa vykonal backtracking ako vždy a jeho následné zapísanie prebehlo jedným forom odzadu, pretože backtracking vlastne vypíše cestu akoby odzadu. V prípade nedostupnosti princeznej, draka to vráti -1. Na konci uvoľním pamäť všetkých alokovaných dynamických pamätí.

Pokračovanie zachran_princezne

Po vykonaní funkčnej časti algoritmu, kedy už máme zapísanú cestu draka. Sa posúvame do nižšej časti funkcie, kde nasleduje 5 scenárov vyplývajúcich zo zadania. Jeden scenár je že bude 1 princezná, druhý, že 2, tretí 3, štvrtý 4 a piaty, že princezien bude 5. Ukážeme si jeden ukážkový, kde ich bude 5. Prvá vec, ktorú urobíme je to, že prepíše permutácie. To znamená, že ak je tam {1,2,3,4,5} tak to prepíšeme na reálne indexy princezien, ktoré sme si zistili pri vytváraní matíc. Následne nastavíme minimálnu cestu na INFINITY, čiže ešte žiadna, tempCestu nastavíme na 0 a premennú pom pomôžeme na uloženie toho, ktorý riadok / resp. scenár v permutácii bol najvýhodnejší.

Urobíme for, ktorý urobí dijkstru pre každú permutáciu, čiže $120 * 5 = 600 \times$ dijkstra. Bez zapisovania cesty.

V ďalšom kroku urobíme ešte 5x dijkstru ale s rozdielom toho, že už vieme najlepší scenár a povolíme zapisovanie cesty, ktorú napojíme na cestu draka. Toto nie je nutný krok, dalo by sa to zjednodušiť napríklad tým, že by som si pri každom dijkstrovi ukladal cestu, čiže by som mal 120 celých ciest alebo 600 malých, takto mám iba jednu veľkú, resp. 5 malých. Najkratšia cesta sa zapíše do hodnoty pointera dlzka_cesty.

```
if(p%60000==5)
{
    for(i=0;i<120;i++)
    {
        for(j=0;j<5;j++)
        {
            if( perm5[i][j]==1 && prin[0][0]==1 && prin[0][1]!=-1 )perm5[i][j]=prin[0][1];
            if( perm5[i][j]==2 && prin[1][0]==1 && prin[1][1]!=-1 )perm5[i][j]=prin[1][1];
            if( perm5[i][j]==3 && prin[2][0]==1 && prin[2][1]!=-1 )perm5[i][j]=prin[2][1];
            if( perm5[i][j]==4 && prin[3][0]==1 && prin[3][1]!=-1 )perm5[i][j]=prin[3][1];
            if( perm5[i][j]==5 && prin[4][0]==1 && prin[4][1]!=-1 )perm5[i][j]=prin[4][1];
        }
    }

    int minCesta = INFINITY;
    int tempCesta = 0;
    int pom = -1;

    for(i=0;i<120;i++)
    {
        tempCesta = 0;
        tempCesta += algoritmus(mapaMATRIXfinal,x-1,drak,perm5[i][0],poleHciek,h%100000,cestovnaMapa,&indexCMapy,0,prin,p%60000);
        tempCesta += algoritmus(mapaMATRIXfinal,x-1,perm5[i][0],perm5[i][1],poleHciek,h%100000,cestovnaMapa,&indexCMapy,0,prin,p%60000);
        tempCesta += algoritmus(mapaMATRIXfinal,x-1,perm5[i][1],perm5[i][2],poleHciek,h%100000,cestovnaMapa,&indexCMapy,0,prin,p%60000);
        tempCesta += algoritmus(mapaMATRIXfinal,x-1,perm5[i][2],perm5[i][3],poleHciek,h%100000,cestovnaMapa,&indexCMapy,0,prin,p%60000);
        tempCesta += algoritmus(mapaMATRIXfinal,x-1,perm5[i][3],perm5[i][4],poleHciek,h%100000,cestovnaMapa,&indexCMapy,0,prin,p%60000);
        if(cestaDraka + tempCesta < minCesta){
            minCesta=cestaDraka+tempCesta;
            pom = i;
        }
    }

    algoritmus(mapaMATRIXfinal,x-1,drak,perm5[pom][0],poleHciek,h%100000,cestovnaMapa,&indexCMapy,1,prin,p%60000);
    algoritmus(mapaMATRIXfinal,x-1,perm5[pom][0],perm5[pom][1],poleHciek,h%100000,cestovnaMapa,&indexCMapy,1,prin,p%60000);
    algoritmus(mapaMATRIXfinal,x-1,perm5[pom][1],perm5[pom][2],poleHciek,h%100000,cestovnaMapa,&indexCMapy,1,prin,p%60000);
    algoritmus(mapaMATRIXfinal,x-1,perm5[pom][2],perm5[pom][3],poleHciek,h%100000,cestovnaMapa,&indexCMapy,1,prin,p%60000);
    algoritmus(mapaMATRIXfinal,x-1,perm5[pom][3],perm5[pom][4],poleHciek,h%100000,cestovnaMapa,&indexCMapy,1,prin,p%60000);

    *dlzka_cesty = minCesta;
}
```

Posledná časť programu obsahuje cyklus, ktorý bude vytvárať pole, ktoré bude obsahovať cestu podľa žiadaného formátu. Obsahuje ešte logickú časť, v ktorej premáva duplikáty, pretože v dijkstrovi to tam zaráta začiatok a koniec vždy dvakrát. Preto sa jedenkrát odstráni aby tam nebol navyše, čo by bola chyba. Potom teda navrátim backtracknutú otočenú cestu a tú vypíše testovací main. Zo zadania mi nebolo jasné, či som mal nejako špeciálne ošetrovať čas t, pretože ten vyhodnocovala testovacia funkcia, či to popolvár stihol alebo nie. Tak som to neošetroval a funkcia vždy vyráta cestu k drakovi a následne k princeznám, tú najlepšiu a najvýhodnejšiu. Ku koncu funkcie uvoľním pamäť všetkých dynamických pamätí. Snažil som sa niektoré veci popísať priamo v kóde ale väčšinu popisu som spravil priamo v dokumentácii, pretože to bolo dosť dlhé a neprišlo mi vhodné opisovať každý riadok kódu, tak som to opísal ako jednotlivé cykly. Ošetril som taktiež viaceré okrajové prípady ako je viac drakov ako 1, viac princezien ako 5, nedostupné princezné, nedostupný drak a taktiež neplatné mapy.

Testovanie

Snažil som sa urobiť, čo najukážkovejšie testy, kde vidno, že to hľadá správne, taktiež, že sú ošetrené okrajové veci, ako nedostupnosť princezien a tomu podobné. U niektorých som nedal výstup z dôvod, že mal napríklad desiatky riadkov a ani v stĺpoch to nemalo moc význam uvádzať ich.

Časová aj pamäťová zložitosť môjho algoritmu je približne $O(\text{početVrcholov}^2)$. Týmto som si nie úplne istý, pretože sme ešte nepreberali ako správne vypočítať zložitosti jednotlivých algoritmov.

Test1 (menší)

Vstup:

6 6 100

CCHCCH

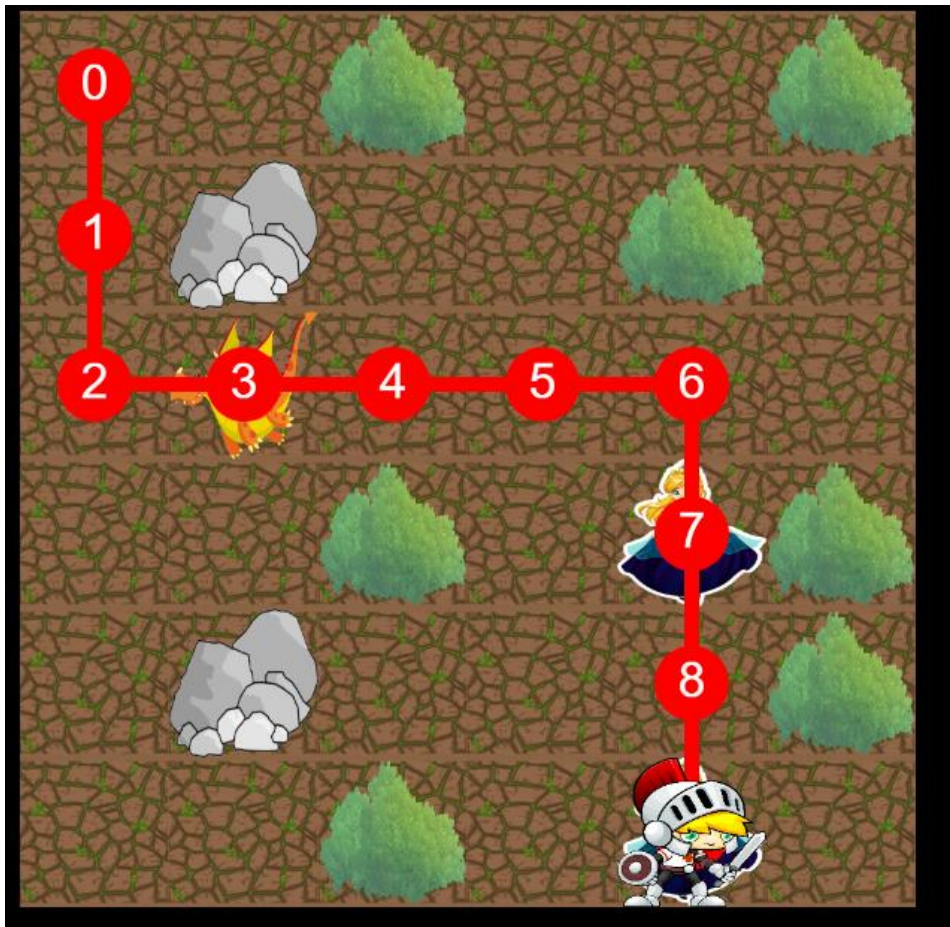
CNCCHC

CDCCCC

CCHCPH

CNCCCH

CCHCPC



Výstup:

0 0

0 1

0 2

1 2

2 2

3 2

4 2

4 3

4 4

4 5

10

Test2 (menší)

Vstup:

6 6 100

CCHCCH

CNCCHC

CDCCCC

CPHCPH

CNCCCH

CCHCPC



Výstup

0 0

0 1

0 2

1 2

1 3

2 3

3 3

4 3

4 4

4 5

1 1

Test3

Vstup:

8 8 30

PCCCCPCC

CCCCCCCC

CCCCCHHH

CCCNHHH

CCCNDNCC

CCCCCCCC

CCCCCCCC

CCCCCCCC



Výstup:

0 0	3 5
1 0	2 5
2 0	2 4
2 1	2 3
2 2	2 2
2 3	2 1
2 4	2 0
2 5	1 0
3 5	0 0
4 5	1 0
4 4	2 0
4 5	3 0

Test 4

Vstup:

```
16 8 30
PCCCCPCC
CCCCCCCC
CCCCCHHH
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
PCCCCPCC
CCCCCCCC
CCCCCHHH
CCCNHHH
CCNDNCC
CCCCCCCC
CCCCCCCC
CCCCCCCC
```



Výstup je príliš dlhý. Vložil som ho do simulátora, obrázok je vygenerovaný z výstupu môjho programu a vstupu.

Test 6 (5 princezien, rohové pozície)

Vstup

8 8 30

PCCCCCP

CCCCCCCC

CCCCCHHH

CCCNHHH

CCCNDNCP

CCCCCCCC

CCCCCCCC

PCCCCCP



Výstup

0 0	4 5	7 1
1 0	5 5	7 0
2 0	6 5	6 0
2 1	7 5	5 0
2 2	7 6	4 0
2 3	7 7	3 0
2 4	7 6	2 0
2 5	7 5	1 0
3 5	7 4	0 0
4 5	7 3	0 1
4 4	7 2	0 2

Test 7 (nedostupná princezná)

Vstup:

16 8 30

PCCCNCCP

CCCNCCC

CCCNHHH

CCCCNNN

CCCCCCC

CCCCCCC

CCCCCCC

CCCCCCC

PCCCPCC

CCCCCCC

CCCCHHH

CCCNHHH

CCNDNCC

CCCCCCC

CCCCCCC

CCCCCCC

Tomáš Rafaj – Popolvár – Zadanie 3



Test 8 (málo času)

Vstup

16 8 7

PCCCCCP

CCCCNCCC

CCCCNHHH

CCCCCINN

CCCCCCCC

CCCCCCCC

CCCCCCCC

CCCCCCCC

PCCCCPCC

CCCCCCCC

CCCCCHHH

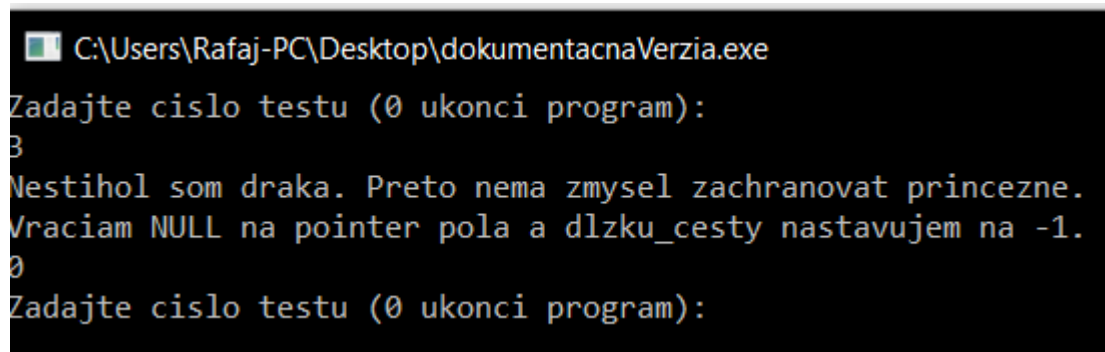
CCCCNHHH

CCCNDNCC

CCCCCCCC

CCCCCCCC

CCCCCCCC

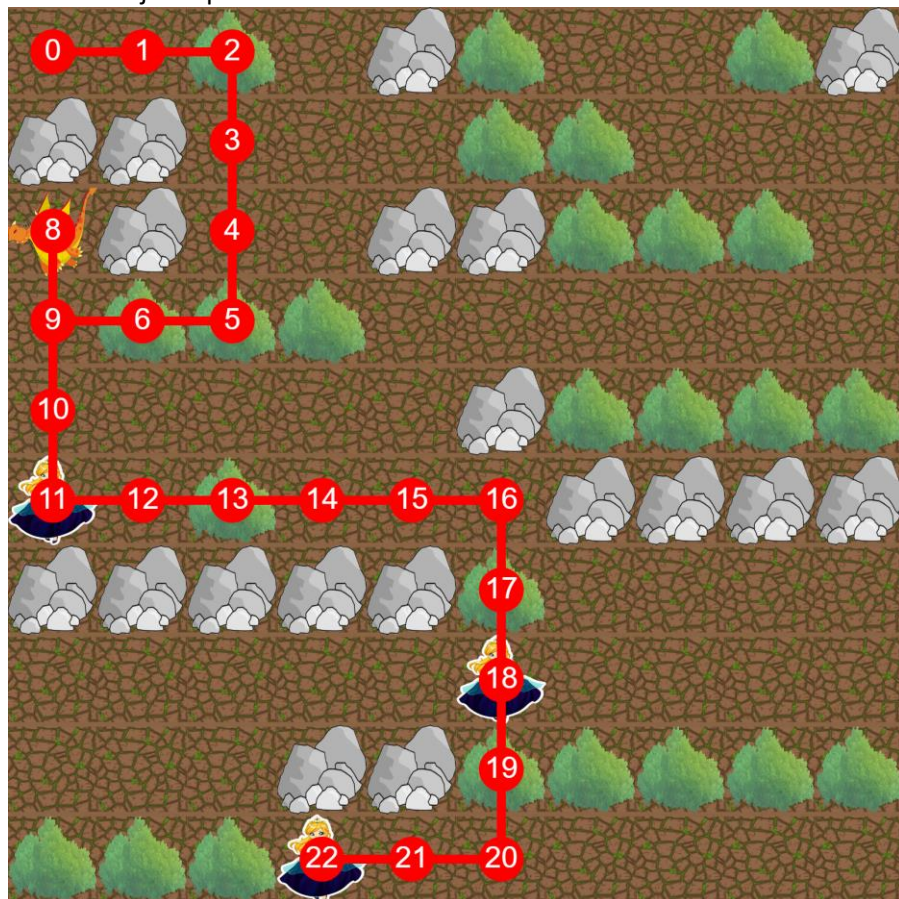


```
C:\Users\Rafaj-PC\Desktop\dokumentacnaVerzia.exe
Zadajte cislo testu (0 ukonci program):
3
Nestihol som draka. Preto nema zmysel zachranovat princezne.
Vraciam NULL na pointer pola a dlzku_cesty nastavujem na -1.
0
Zadajte cislo testu (0 ukonci program):
```

Test 9 (ukážkový test)

Vstup ukážkový test zo zadania3_testovanie.

Tomáš Rafaj – Popolvár – Zadanie 3

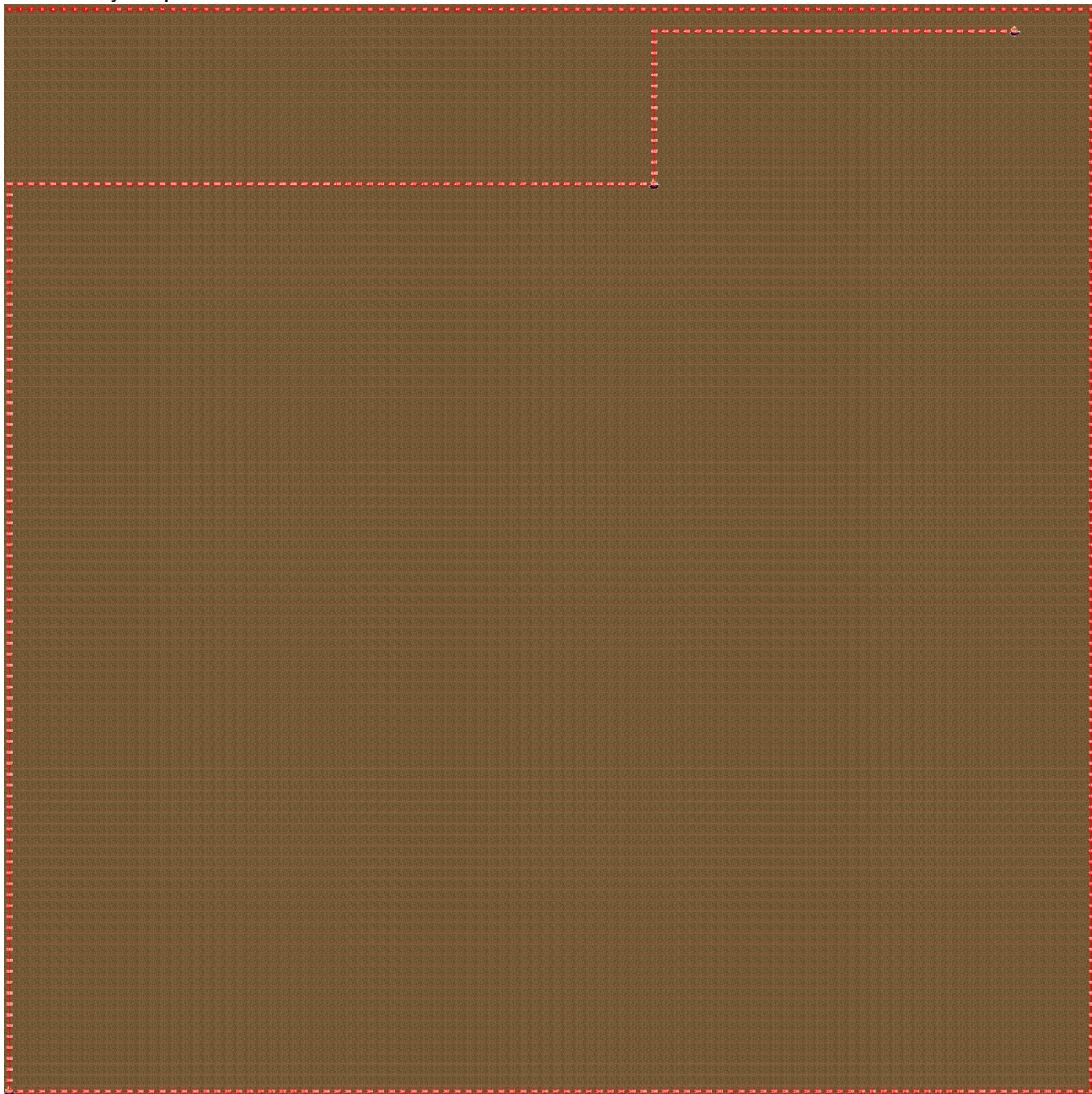


Výstup:

0 0	1 5
1 0	2 5
2 0	3 5
2 1	4 5
2 2	5 5
2 3	5 6
1 3	5 7
0 3	5 8
0 2	5 9
0 3	4 9
0 4	3 9
0 5	2 9

Test10 (maximálna mapa)

Vstup: Obsahuje testovanieVlastne10.txt



Výstup je príliš dlhý. Vložil som ho do simulátora, obrázok je vygenerovaný z výstupu môjho programu a vstupu.