Martin Mihalovič
Pondelok 18:00, ID: 96969

# Dátové štruktúry a algoritmy

## Zadanie č. 3 – Popolvár

**Martin Mihalovič**

Martin Mihalovič
Pondelok 18:00, ID: 96969

## Použité algoritmy

V mojom projekte som použil algoritmy Dijkstru, minimálnu binárnu haldu a haldový algoritmus („Heap algoritmus“) na vytvorenie permutacií. Tieto algoritmy som využival v štruktúre grafu, ktorý som reprezentoval ako „adjacency list“.

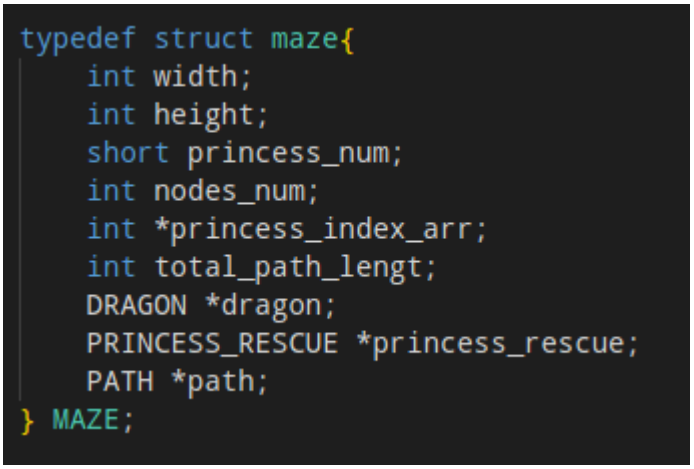
„Adjacency list“, je reprezentácia grafu, kde každé poličko v bludisku, kde vedie cesta, je „path root“ v zozname cesty. Ak toto poličko cesty má susedov, kde taktiež vedie cesta, tak od „path root-a“ vedieme spájaný zoznam susedov. Keďže dané políčko v bludisku („path root“) môže mať maximálne štyroch susedov (severne, južne, západne alebo východne od seba), tak maximálna veľkosť spájaného zoznamu je päť prvkov.

Dijsktrov algoritmus slúži na nájdenie najlacnejšie nájdenie cesty, počas ktorého využivam minimálnu binárnu haldu, kde si ukládam hodnoty ciest ešte k neobjaveným vrcholom v grafu (v pripáde ak hodnota nie je infinity).

Haldový algoritmus využívam pri vytváraní permutácií poradia navštívenia princezien v bludisku. Počet permutácií bude priamoúmerne teda závisieť od počtu princezien a teda bude to n faktoriál.

## Bludisko

Kód som sa snažil čo najviac štrukturalizovať a modularizovať. Myšlienka od začiatku programovania tohto projektu bola, si držať všetky dáta pokope, najlepšie v jednej štruktúre. Logickou implikáciou som postupoval k tomu, že som si vytvoril štruktúru bludiska, kde som si držal môj projektový svet (bludisko). V tejto štruktúre „maze“ uchovávam výšku a šírku mapy, počet princezien a vrcholov, pole indexov princezien, celkovú výslednú dĺžku ale tiež ďalšie štruktúry draka, záchrany pricezien a samotnej cesty.



```
typedef struct maze{
    int width;
    int height;
    short princess_num;
    int nodes_num;
    int *princess_index_arr;
    int total_path_lengt;
    DRAGON *dragon;
    PRINCESS_RESCUE *princess_rescue;
    PATH *path;
} MAZE;
```

Martin Mihalovič
Pondelok 18:00, ID: 96969

Štruktúra cesta je moja reprezentácia grafu. Každý graf ma toľko ciest koľko je vrcholov cesty a každý tento vrchol je „path root“ v danej ceste. Každá štruktúra cesty si uchováva informáciu, či už bola objavená a koľko stojí.

Cena každej tejto cesty na začiatku je INFINITY (v mojom pripáde definovaná konštanta s hodnotou 10000) ak daná cesta neobsahuje štartovací „path root“. Každý vrchol cesty („path node“) má svoje id (pre účelý rýchlejšej identifikácie v algoritme), hodnotu (1 ak je lesný chodník, princezná alebo drak a 2 ak je to strmá šikmina), pozíciu kde sa v mape nachádza a nakoniec referenciju na ďalší vrchol v spájanom zozname.

Štruktúra „source path“ uchováva počet vrcholov na ceste k danému a index posledného. Index potrebujem pre účely dijkstra algoritmu.

```c
/* root in adjacency list */
typedef struct path_node{
    int id;
    int cost;
    COORDINATES position;
    struct path_node *next;
} PATH_NODE;


typedef struct source_path{
    int index_of_src_path_root;
    int num_of_src_path_nodes;
} SOURCE_PATH;


/* Path in adjacency list */
typedef struct path{
    int cost;
    int known;
    SOURCE_PATH *source_path;
    PATH_NODE *path_root;
} PATH;
```

Štruktúra draka v štruktúre bludiska uchováva maximálny čas t, ktorý som dostal ako argument do funkcie „zachrán_princeznu“, index v grafe a pozíciu (x,y) v grafe.

```c
typedef struct dragones_{
    int t;
    int index;
    COORDINATES position;
} DRAGON;
```

Martin Mihalovič
Pondelok 18:00, ID: 96969

Posledná štruktúra v bludisku je „princess rescue“. S touto štruktúrov pracujem primárne pri „brute force“ alebo inými slovami permutačnom algoritme „Heap“ algoritmus.  Držím si tu permutácie ciest navštívenia princezien, cesty samotných permutácií, čas „t“ koľko daná permutácia cesty trvá a aktuálny počet zachránených princezien.

```
typedef struct princess_rescue{
    int t;
    int *permutation_of_princess_indexes;
    int **rescue_path_of_princess_permutation;
    int num_princess_rescue_path;
} PRINCESS_RESCUE;
```

### Graf – „adjacency list“ reprezentácia

Pri volaní funkcie „zachran_princezne“ prvotne inicializujem bludisku a samotný graf. Následne prechádzam mapu a načítavam vrcholy do grafu, spôsobom ako som opísal vyššie v dokumentácii - pozriem sa na akom políčku som, vytvorím vrchol (path root) a pozriem sa na susedov vo všetkých štyroch svetových stranách.

```
for (y = 0; y < maze->height; y++){
    for (x = 0; x < maze->width; x++){
        if (mapa[y][x] == FOREST_PATH || mapa[y][x] == SLOW_WAY || mapa[y][x] == PRINC || mapa[y][x] == DRAG){

            new_node = init_path_node(
                y * 100 + x,
                mapa[y][x] == SLOW_WAY
                    ? SLOW_PATH_VALUE
                    : FOREST_PATH_VALUE,
                y, x);

            /* Saving index of the princess */
            if(mapa[y][x] == PRINC)
                maze->princess_index_arr[maze->princess_num++] = maze->nodes_num;

            /* Saving dragon as a struct */
            else if(mapa[y][x] == DRAG){
                drak = init_dragon(t, maze->nodes_num, y, x);
                maze->dragon = drak;
            }

            maze = create_vertex(maze, new_node, mapa, maze->nodes_num);
            maze->nodes_num++;
        }
    }
}
```

```
/* smer - Vychod */
neighboor_value = check_node_neighbors(maze, map, x + 1, y);
maze = neighboor_value != FALSE ? add_neighboor_to_path(maze, new_node, neighboor_value, index, y, (x + 1))
                                : maze;

/* smer - Zapad */
neighboor_value = check_node_neighbors(maze, map, x - 1, y);
maze = neighboor_value != FALSE ? add_neighboor_to_path(maze, new_node, neighboor_value, index, y, (x - 1))
                                : maze;

/* smer - Sever */
neighboor_value = check_node_neighbors(maze, map, x, y + 1);
maze = neighboor_value != FALSE ? add_neighboor_to_path(maze, new_node, neighboor_value, index, (y + 1), x)
                                : maze;

/* smer - Juh */
neighboor_value = check_node_neighbors(maze, map, x, y - 1);
maze = neighboor_value != FALSE ? add_neighboor_to_path(maze, new_node, neighboor_value, index, (y - 1), x)
                                : maze;
```

Po úspešnom načítaní mapy, máme graf v reprezentácii „adjacency“ list-u  a predpripravený pre účely algoritmov dijkstra a minimálne binárnej haldy.

```
i:0 id: 0 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [0, 0] -> [1, 0]
i:1 id: 2 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [0, 2] -> [1, 2]
i:2 id: 100 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [1, 0] -> [1, 1] -> [0, 0]
i:3 id: 101 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [1, 1] -> [1, 2] -> [1, 0]
i:4 id: 102 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [1, 2] -> [1, 3] -> [1, 1] -> [0, 2]
i:5 id: 103 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [1, 3] -> [1, 2] -> [2, 3]
i:6 id: 203 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [2, 3] -> [3, 3] -> [1, 3]
i:7 id: 300 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [3, 0] -> [3, 1]
i:8 id: 301 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [3, 1] -> [3, 2] -> [3, 0]
i:9 id: 302 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [3, 2] -> [3, 3] -> [3, 1]
i:10 id: 303 path cost: 10000 known: 0 src path node: -1 source nodes 0,  [3, 3] -> [3, 2] -> [2, 3]
```

## Dijkstrov algoritmus

Algoritmus dijkstra sa využíva na hľadanie najlacnejšej cesty. Ako argument potrebný pre vykonanie tohto algoritmu je počiatočný vrchol grafu, z ktorého algoritmus dijkstra vykonávam.

Prvotne algoritmus prechádza priamych susedov štartovacieho vrcholu. Tieto susedné vrcholy už mám v linked liste „path roota“ uložené. Prechádzam tieto vrcholy a ohodnocujem cestu, podľa toho akú majú hodnotu dané „path node-i“ (teda, či sú lesná cesta/princezná alebo šikmá, spomalená cesta).

```c
/* Prejde linked list od pathroota horizontalne */
void find_and_update_neighboor(MAZE *maze, HEAP *heap, int index){

    /* prvy sused od path root-a*/
    PATH_NODE *current = maze->path[index].path_root->next;

    /* path cost momentalny */
    int root_path_cost = maze->path[index].cost;

    /* prechadzam vsetkych susedov od path-root-a */
    while(current != NULL){
        change_path_cost(maze, current, heap, root_path_cost, index);
        current = current->next;
    }
}
```

Tieto hodnoty následne vkladám do binárnej haldy ak aktuálna cena cesty, je nižšia ako predchádzajúca . Na začiatku majú všetky hodnotu infinity a nie sú v min. halde, teda ak ešte dané ohodnotenie cesty je infinity a nie je objavené, tak vložím ohodnotenie do min. haldy. Ak hodnota už nie je infinty, nová možná hodnota je nižšia ako pôvodná a taktiež daný „path root“ ešte nie je objavený, tak zmením hodnotu cesty v halde aj v grafe za nižšiu.

Všetkých susedov aktualneho „path roota“ prejdem a ak vyhovujú podmienkam tak vložím do min. haldy.

```c
/* Ak dany path root este nie je objaveny a jeho path cost je vyssia ako aktualna */
if(maze->path[i].known == FALSE && path_node->cost + root_path_cost < maze->path[i].cost){

    /**
     * Ak je predchadzajuci cost infinity, to znamena ze v halde este nie je zaznam toho cost
     * ak nie je infinity zanmena z v halde uz je zaznam a preto ho treba zmenit aby nevznikali duplikaty
    */
    heap = maze->path[i].cost == INFINITY ? insert_heap_node(heap, i, path_node->cost + root_path_cost)
                                          : change_for_cheaper_cost(heap, i, path_node->cost + root_path_cost);

    /* updatujem ho na lacnejsi cost aj v grafe */
    maze->path[i].cost = path_node->cost + root_path_cost;

    /* zmenim src node v grafe */ int source_path::index_of_src_path_root
    maze->path[i].source_path->index_of_src_path_root = root_path_index;
}
```

Po vložení všetkých susedov počiatočného/štartovacieho vrcholu, si vyberiem z min haldy vrchol s najmenším ohodnotením cesty, teda root v min halde. Tento vrchol označím ako objavený. Algoritmus iterujem, tým že nájdem všetkých susedov, ktorý ešte nie sú objavený, aktualizujem im ohodnotenie cesty ak vyhovujem podmienkam. Algoritmus končí vtedy ak už z haldy nemám čo vybrať a teda halda je NULL.

```c
/* vyberam kym je nieco v halde */
while((heap = push_and_pop_min(heap, index)) != NULL){

    /* funkcia push and pop min mi cez premennu index updatuje index min cost-u cesty a ten oznacim ako known true */
    set_known_root_node(maze, *index);

    /* najdem susedov path roota ktory ma priebezny najmensi cost */
    find_and_update_neighboor(maze, heap, *index);
}
```

Ak while cyklus skončí a teda halda ostane prázdna, tak linearné ešte prejdem pole „path rootov“. Ak niektorý vrchol ostal neobjavený, znamená, že k nemu nevedie cesta. V tomto prípade musím skontrolovať či medzi neobjavenými vrcholmi nie je aj princezná alebo drak. Ak by medzi nimi bol vrchol s princeznou alebo drakom, znamenalo by to, že nevedie k nim cesta a funkcia „zachran_princezne“ vrati NULL s dlzkou cesty nula.

Vo funckii, okrem kontroly aktualizujem informácie o „source path root“ vrcholoch.

```c
int check_and_update_src_paths(MAZE *maze, int starting_index){
    int index_counter, i, j;

    for (i = 0; i < maze->nodes_num; i++){
        if (maze->path[i].known == FALSE){
            for (j = 0; j < maze->princess_num; j++){

                /* Kontrola ci sa jedna o princeznu - ak ano tak je zle a neviem sa k nim dostat */
                if(i == maze->princess_index_arr[j]){
                    printf("Princezna zablokovana!\n");
                    return 0;
                }
            }

            /* Kontrola ci sa jedna o draka - ak ano tak je zle a neviem sa k nim dostat */
            if(i == maze->dragon->index){
                printf("Drak zablokovany!\n");
                return 0;
            }
        } else{

            /* pripocitam najdeny node ako source */
            maze->path[i].source_path->num_of_src_path_nodes++;
```

Martin Mihalovič
Pondelok 18:00, ID: 96969

**Minimálna binárna halda**

Minimálna halda mi v programe vykonáva funkcionalitu prioritného radu. To znamená, že bez lineárneho prehliadania O(n) pri vyberáni najmenšej hodnoty z pola ohodnotení ciest, viem hneď O(1) vybrať najmenšiu hodnotu z haldy, pretože je ako „root“ v halde.

V minimálnej binárnej halde platí pravidlo, že rodič nemôže byť väčší ako dieťa v strome.  Taktiež platí, že rodič má dve deti a k ich indexom sa viem v binárnej halde dostať pomocou výpočtu: (index_rodiča x 2) + 1, ľavý potomok a (index_rodiča x 2) + 2, pravý potomok.

Z potomka sa viem dostať k indexu rodiča pomocou výpočtu (index_potomka-1)/2.

Vladanie do binárnej haldy riešim tak, že prvok vložim prv na posledné miesto v poli haldy.

```
HEAP *insert_heap_node(HEAP *heap, int index, int path_cost){
    HEAP_NODE *heap_node = init_heap_node(index, path_cost);

    /* resolution statement, checking max heap size */
    if(heap->heap_act_size >= heap->heap_max_size){
        printf("No success insert to heap - ochrana proti preteceniu\n");
        return heap;
    }

    int act_index = heap->heap_act_size;

    /* vlozim dany heap_node do pola */
    heap->heap_node[act_index] = *heap_node;
    heap = clean_in_heap(heap, act_index);
    heap->heap_act_size++;

    return heap;
}
```

Avšak po vložení musím upratať/skontrolovať upratanie v halde. Na to mi slúži funkcia „clean_in_haep“ , ktorá sa pozrie na rodiča vloženej hodnoty. Ak je hodnota rodiča väčšia, tak ich v poli vymením. Takto pokračujem až kým hodnota rodiča bude menšia alebo rovná hodnote čerstvo vloženého potomka.

```
/* reordering and cleaning heap in way of min heap rules */
HEAP *clean_in_heap(HEAP *heap, int i){
    HEAP_NODE *parent;

    while(i != 0 && heap->heap_node[((i - 1) / 2)].value > heap->heap_node[i].value){
        /* Vymenim ich v poli */
        swap_heap_nodes(&heap->heap_node[i], &heap->heap_node[((i - 1) / 2)]);
        i = (i - 1) / 2;
    }

    return heap;
}
```

Ak už ohodnotenie cesty v halde sa nachádza a aktuálny možný je menší ako predošlí, tak haldu prejdem lineárne kým nenájdem rovnaký index ohodnotenia cesty, ten aktualizujem za novšiu, výhodnejšiu cenu cesty a znova upracem v halde od indexu editovaného ohodnotenia cesty.

```c
/* fn for change cost which already is for smaller one */
HEAP *change_for_cheaper_cost(HEAP *heap, int index, int new_path_cost){
    int i;

    for (i = 0; i < heap->heap_act_size; i++){
        if(heap->heap_node[i].index == index){
            heap->heap_node[i].value = new_path_cost;
            heap = clean_in_heap(heap, i);

            return heap;
        }
    }

    printf("change_for_cheaper_cost - nieco je zle lebo som ho nenasiel v halde\n");
}
```

Pri vyberaní najmenšieho prvku z min. haldy využívam funckiu „push_and_pop_min“, ktorá sa najskôr pozrie, či existuje ešte nejaké ohodnotenie cesty v halde. Ak nie, tak uvoľní pole haldy, a dijkstra algoritmus pokračuje.

Ak sa tu nachádza presne jedna hodnota, tak zmením veľkosť na nulu a upratovať nemusím, pretože v halde už nič neostalo.

Ak počet prvkov v halde je väčší ako jeden, tak znížim počet prvkov, ako najmenší (root) nastavím prvok z posledného indexu, teda jeden z najväčších a upracem z hora na dol ( od roota až po listy ).

```c
/* vyberie najmensi cost z haldy (teda root) a pop-ne ho, nalsedne uprace haldu */
HEAP *push_and_pop_min(HEAP *heap, int *path_index_of_min){
    *path_index_of_min = get_path_index_of_min(heap);

    if(heap->heap_act_size <= 0){
        free(heap);
        return NULL;
    }

    if (heap->heap_act_size == 1){
        heap->heap_act_size--;
        return heap;
    }

    heap->heap_act_size--;
    heap->heap_node[0] = heap->heap_node[heap->heap_act_size];
    heapifyTtB(heap, 0);

    return heap;
}
```

Martin Mihalovič
Pondelok 18:00, ID: 96969

Funkcia „heapifyTtB“ uprace pole haldy, tak aby platili pravidlá minimálnej haldy. Teda pozrie sa na pravé a ľavé dieťa. Ak hodnota oboch týchto detí je menšia ako hodnota rodiča, tak rodiča sa vymení v poli s dieťatom s nižšou hodnotou. Takto docielim, aby rodič bol vždy najmenší a teda aj „root“ bol najmenší.

Podobne iba ak ľavé dieťa je menšie ako rodič, tak vymieňam v poli rodiča a ľavé dieťa. S pravým dieťaťom je to obdobne, len výmena následne prebieha medzi rodičom a pravým potomkom.

```
/**
* left and right parent as well are optionall (have a bigger cost),
* choose with smaller cost between tham and replace with child
*/
if((left < heap->heap_act_size && heap->heap_node[index].value > left_child->value) &&
    (right < index && heap->heap_node[index].value > right_child->value)){

    swap_heap_nodes(&heap->heap_node[index], left_child->value < right_child->value
                                            ? &heap->heap_node[left]
                                            : &heap->heap_node[right]);

    heapifyTtB(heap, left_child->value < right_child->value
                    ? left
                    : right);
}

/**
* left parent have a bigger cost, replace with child
*/
if(left < heap->heap_act_size && heap->heap_node[index].value > left_child->value){
    swap_heap_nodes(&heap->heap_node[index], &heap->heap_node[left]);
    heapifyTtB(heap, left);
}

/**
* right parent have a bigger cost, replace with child
*/
if(right < heap->heap_act_size && heap->heap_node[index].value > right_child->value){
    swap_heap_nodes(&heap->heap_node[index], &heap->heap_node[right]);
    heapifyTtB(heap, right);
}
```

Martin Mihalovič
Pondelok 18:00, ID: 96969

## Heap algoritmus

Heap algoritmus je prebraný a upravený na vytvorenie rôznych permutácií, v akom poradí možno navštíviť princezné.[1]

Pre každú túto permutáciu následne vykonám algoritmus najkratšej cesty a vyberiem z premutácii cestu navštívenia všetkých princezien s najlacnejším ohodnotením.

```
/**
 * Generating permutation using Heap Algorithm
 * zdroj: https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/
*/
void heapPermutation(MAZE *maze, int a[], int size, int index) {
    int static counter = 0;
    // if size becomes 1 then prints the obtained
    // permutation
    if (size == 1) {
        maze = init_princess_rescue(maze, a, maze->princess_num, counter++);
        return;
    }

    for (int i=0; i<size; i++) {
        heapPermutation(maze, a, size-1, index);

        // if size is odd, swap first and last
        // element
        if (size%2==1)
            swappiness(&a[0], &a[size-1]);

        // If size is even, swap ith and last
        // element
        else
            swappiness(&a[i], &a[size-1]);
    }
}
```

## Testovanie

Moje testovanie pozostáva z 29 testov, ktorý každý je zameraný na niečo iné. Testy sú rozdelené do piatich kategórií – ukážkový test, malé, stredné, veľké testy a testy krajných prípadov.

Každá mapa je uložená v textovom súbore v priečinku „testing_maps“.

```
printf("======= Vitajte v testovacom prostredi algoritmu dijkstra, minimalnej haldy a heap algoritmu! =======\n");
printf("Prosim vyberte si test (cislo 0-1):\n");
printf("\t- 0. Test ukazkovy 10x10 - 3P\n");
printf("\t- 1. Test maly 4x4 - 1P\n");
printf("\t- 2. Test maly 5x4 (najlacnejsia cesta rozhodnutie)- 3P\n");
printf("\t- 3. Test maly 6x6 - 5P\n");
printf("\t- 4. Test maly 7x100 - 2P\n");
printf("\t- 5. Test maly 7x100 - 4P\n");
printf("\t- 6. Test stredny 20x20 - 1P\n");
printf("\t- 7. Test stredny 20x20 - 5P\n");
printf("\t- 8. Test stredny 21x21 - 2P\n");
printf("\t- 9. Test stredny 21x21 - 4P\n");
printf("\t- 10. Test stredny 25x50 - 3P\n");
printf("\t- 11. Test stredny 25x50 - 5P\n");
printf("\t- 12. Test stredny 15x100 - 1P\n");
printf("\t- 13. Test stredny 15x100 - 4P\n");
printf("\t- 14. Test velky 30x100 - 3P\n");
printf("\t- 15. Test velky 30x100 - 5P\n");
printf("\t- 16. Test velky 50x50 - 2P\n");
printf("\t- 17. Test velky 50x50 - 4P\n");
printf("\t- 18. Test velky 100x100 - 3P\n");
printf("\t- 19. Test velky 100x100 - 5P\n");
printf("\t- 20. Test krajnych pripadov 1x1 - ziadna princezna ani drak\n");
printf("\t- 21. Test krajnych pripadov 1x1 - ziadna princezna iba drak\n");
printf("\t- 22. Test krajnych pripadov 1x2 - jedna princezna a drak\n");
printf("\t- 23. Test krajnych pripadov 1x3 - drak a zablokovana princezna verticalne\n");
printf("\t- 24. Test krajnych pripadov 2x4 - jedna princezna a zablokovany drak\n");
printf("\t- 25. Test krajnych pripadov 2x7 - v bludisku sa nenachadza ziadna princezna\n");
printf("\t- 26. Test krajnych pripadov 15x15 - 2 princezne a drak zablokovane horizontalne\n");
printf("\t- 27. Test krajnych pripadov 7x8 - 4 princezne a ziadny drak\n");
printf("\t- 28. Test krajnych pripadov 5x5 - nepriechodna sachovnica k princeznej\n");
printf("\t- 29. Nechcem uz testovat\n");
```

### Malé testy a ukážkový test

Princíp malých testov som využíval primárne na overiteľnosť správnosti výberu „naozaj najlacnejšej“ cesty ale nie len to. V týchto testoch sa nachádzajú mapy, kde je potrebné vykonať permutácie hladania princezien, pretože, ak by sme na hladania všetkých princezien nepozerali ako celok, tak by sme nenašli (alebo program) naozaj najlacnejšiu cestu.

Výsledkom týchto testov bolo primárne sledovanie správnosti výstupov

### Stredné a veľké testy

Stredné a veľké testy som využívaľ na komplikovanejšie scenáre hľadania draka alebo princezien. Takisto som testoval a sledoval rôzne scenáre počtu princezien.

Výsledkom týchto testov bolo sledovanie primárne rýchlosti programu a korektnosti výstupu.

### Testy krajných prípadov

Testy krajných prípadov je deväť rôznych testov, ktoré sú istým spôsobom krajné prípady. Ako príklad uvediem, že buď k drakovi nevedie cesta, jednej k princezien, nenachádza sa v mape drak alebo princezná alebo v mape je iba jedno políčko a to drak…

Výsledkom testovania týchto scenárov bolo sledovanie, či programy nepadnú a vrátia NULL a dĺžku cesty nula ak nie je v mape drak, princezná alebo niektorému z nich nevediet cesta...

**Štatistika kódu bez súborov txt (máp)**

- 7 súborov (4 .c, 3 hlavičkové)
- +1450 riadkov kódu
- +50 funkcií
- +35 testovacích máp

**Analýza zložitosti programu**

- dijkstra O(n**2)
- min binárna halda:
    - hladanie O(1)
    - vlozenie prvku O(log n)
    - vymazanie O(log n)
- heap algoritmus (permutácie) O(n!)
- Spustenie algoritmu dijkstra:

[O(n!)*počet_princezien]*[O(n**2)*počet_path_rootov + O(n**2)*2*O(log n)]

**Záver**

Kód som sa snažil urobiť čo najviac modulárny, bez zbytočných globálnych premenných, s ohľadom na rýchlosť a pamäťovú efektivitu. Program bol kompilovaný a testovaný na gcc version 9.2.1 20200130 (Arch Linux 9.2.1+20200130-2).

Môj postup na projekte si môžete pozrieť aj na githube - https://github.com/mihino89/Dijkstra-in-maze

**Zdroje**

[1] `https://www.geeksforgeeks.org/heaps-algorithm-for-generating-permutations/`