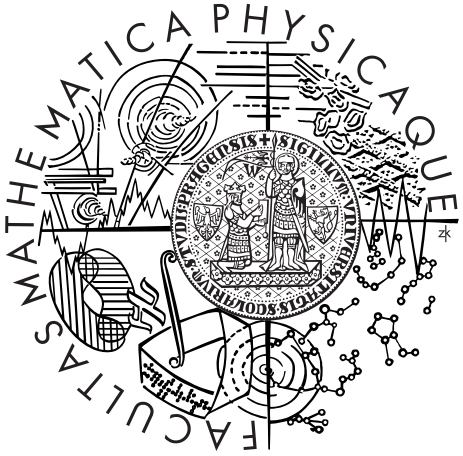
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁRSKA PRÁCA



Tomáš Novella

Grid-Based Path Planning

Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: Mgr. Tomáš Balyo
Študijný program: Informatika
Študijný obor: Obecná informatika

Praha 2013

Rád by som poďakoval predovšetkým vedúcemu práce

Mgr. Tomášovi Balyovi

za vedenie práce, námietky a pripomienky a konzultácie.

Ďalej by som sa rád poďakoval

Mgr. Martinovi Marešovi

za konzultácie ohľadom teórie grafov.

Prehlasujem, že som túto prácu vypracoval samostatne a výhradne s použitím citovaných prameňov, literatúry a ďalších odborných zdrojov.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona a v platnom znení, obzvlášť skutočnosť, že Univerzita Karlova v Prahe má právo na uzavretie licenčnej zmluvy o použití tejto práce ako školského diela podľa §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Názov práce: Grid-Based Path Planning

Autor: Tomáš Novella

Katedra: Katedra teoretické informatiky a matematické logiky

Vedúci bakalárskej práce: Mgr. Tomáš Balyo, Katedra teoretické informatiky a matematické logiky

Abstrakt:

Kľúčové slová: mriežkový graf, problém hľadania najkratšej cesty, algoritmus A*

Title: Grid-Based Path Planning

Author: Tomáš Novella

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Tomáš Balyo, Department of Theoretical Computer Science and Mathematical Logic

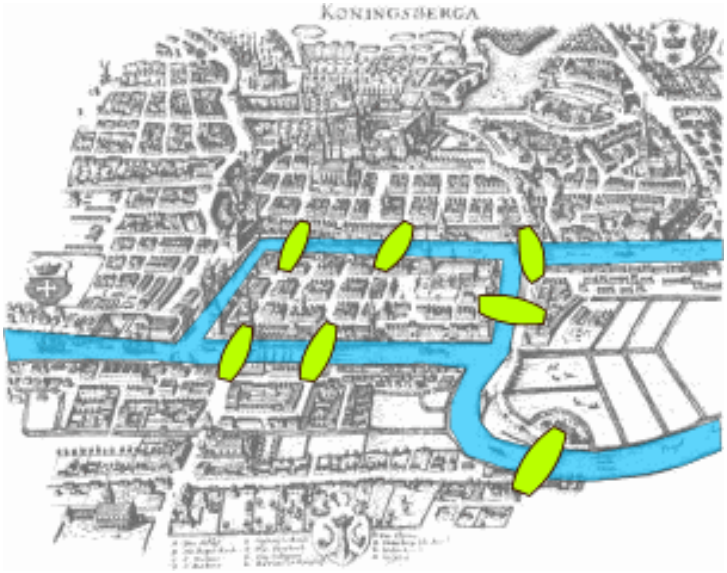
Abstract:

Keywords: grid-based graph, shortest path problem, A* search algorithm

Obsah

Úvod

Slávna Eulerova úloha siedmych mostov v Kaliningrade [1] sa považuje za prvú prácu, ktorá zaviedla teóriu grafov. Úlohou je prejsť po týchto siedmych mostoch tak, aby sme po každom prešli práve raz.



Obr. 1: Sedem mostov v Kaliningrade, `http://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg`

Od tej doby sa využitie teórie grafov značne rozšírilo a v dnešnej dobe patrí medzi významné a rozpracované teórie. V modernej dobe je jedným z jej najdôležitejších problémov hľadanie najkratšej cesty. Najčastejšie sa s nimi stretávame pri plánovaní trasy v GPS navigácii. Medzi najvýznamnešie práce považujeme práce od Dijkstru [2] a Floyd-Warshalla [3].

So začiatkom fenoménu počítačových hier a umelej inteligencie sa do povedomia dostal špeciálny typ grafu – mriežkový graf, využívaný ako herná mapa. V hrách často trebalo nájsť cestu pre počítačom ovládanú postavičku z miesta A do miesta B. Nakoľko je väčšina hier komerčná, algoritmy využívané v hrách boli a sú taktiež komerčné. Dôsledkom toho nie sú verejne publikované a porovnané rôzne prístupy a algoritmy na vyhľadávanie najkratších ciest v mriežkových mapách. A keď už aj sú, tak práce používajú rôzne mapy na bechmarking a teda neexistuje žiadna globálna porovnávacia štúdia týchto prístupov.

Súťaž *Grid-Based Path Planning Competition* [9] sa snaží tento problém vyriešiť tým, že porovnáva rôzne algoritmy na veľkej množine máp použitých v známych počítačových hrách a vyhodnocuje ich úspešnosť v rámci viacerých kategórií.

Cieľom tejto práce je spraviť prehľad doterajších prístupov k tomuto problému a prispieť vlastným algoritmom do sútaže a niekoľkými vylepšeniami k doterajším prístupom hľadania najkratšej cesty na mriežkových grafoch.

V prvej kapitole si zavedieme kľúčové termíny a popíšeme problém formálne. Na konci kapitoly spomenieme súťaž, ktorej sa daný algoritmus zúčastnil. Druhá kapitola je zameraná na vytvorenie prehľadu kľúčových algoritmov používaných

na riešenie problému. V ďalšej kapitole navrhneme vlastné riešenie založené na poznatkoch popísaných v druhej kapitole s pridaním vlastných vylepšení. A v poslednej kapitole toto riešenie porovnáme s dosavadnými.

Práca obsahuje dve prílohy: stručnú užívateľskú a programátorskú dokumentáciu programu *T-maps*, ktorý slúži na grafické zobrazenie mriežkového grafu.

Časť I

Analýza problému

1. Zadanie problému a cieľové požiadavky

1.1 Úvodné definície a značenia

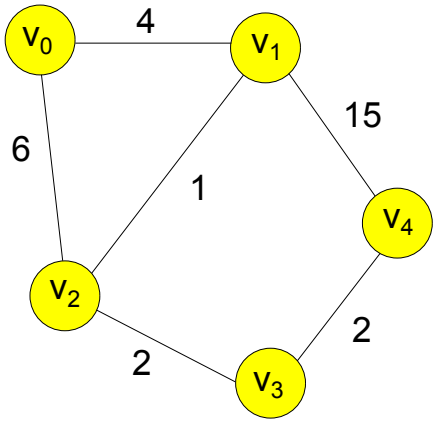
Na začiatok si zaveďme niektoré dôležité pojmy z teórie grafov. Úlohu so všetkými jej špecifikami si ozrejmíme v nasledujúcich podkapitolách.

Definícia 1. *Graf G je usporiadaná dvojica (V, E), kde V označuje množinu vrcholov(vertices) a $E \subseteq V \times V$ označuje množinu hrán (edges). Značíme G = (V, E).*

Poznámka 1. *Hrana je jednoznačne určená dvojicou vrcholov.*

Definícia 2. *Ohodnotený graf (G, w) je graf G spolu s reálnou funkciou (tzv. ohodnotením) $w : E(G) \to \mathbb{R}$, kde w je funkcia, ktorá každej hrane priradí reálne číslo reprezentujúce* dĺžku, *alebo* hodnotu *hrany.*

Ukážka obecného ohodnoteného grafu je na obrázku 1.1.



Obr. 1.1: Ohodnotený graf

Pri hľadaní najkratšej cesty v grafe pracujeme s pojmami, ako sú *cesta* a *najkratšia cesta.*

Definícia 3. *Cesta P z vrcholu v_0 do vrcholu v_n v grafe G je postupnosť $P = (v_0, e_1, v_1, \ldots, e_n, v_n)$, pre ktorú platí $e_i = \{v_{i-i}, v_i\}$ a taktiež $v_i \neq v_j$ pre každé $i \neq j$.*

Všimnime si, že na ceste nenavštívime žiaden vrchol dvakrát a teda cesta neobsahuje kružnice.

Definícia 4. *Dĺžka cesty P z vrcholu v_0 do vrcholu v_n v ohodnotenom grafe (G, w) je súčet dĺžok hrán, ktoré sa na ceste nachádzajú.*

Samozrejme, medzi dvoma vrcholmi môže existovať viacero ciest.

Definícia 5. *Najkratšia cesta P z vrcholu v_0 do vrcholu v_n v ohodnotenom grafe (G, w) je cesta z vrcholu v_0 do vrcholu v_n s najmenšou dĺžkou.*

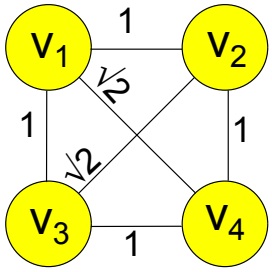
1.2 Mriežkový graf

Keď sme si už zaviedli kľúčové pojmy, prejdime k samotnému zadaniu úlohy. Ako sme už spomínali, problém budeme riešit na tzv. mriežkových grafoch. Čo je mriežkový graf a v čom sa od obecného grafu odlišuje?

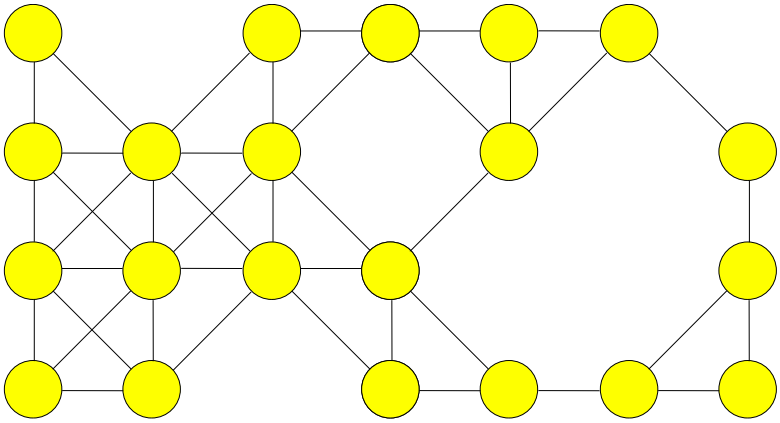
V hernej praxi predstavuje mriežkový graf mapu hracej plochy, s ktorou sa stretávame v najrôznejších hrách, ako je Warcraft, Startcraft, Dragon Age [10] a podobne. Preto niekedy používame pojem *mapa* na označenie mriežkového grafu.

Ide o špeciálny a dosť obmedzený typ grafu. Vizuálne si ho môžme predstaviť ako konečný graf v ktorom sú vrcholy rozostúpené v tvare mriežky a hrana je stále medzi dvojicami susedných vrcholov vo všetkých ôsmych smeroch. Dĺžka vodorovnej alebo zvislej hrany je 1 a dĺžka šikmej hrany je $\sqrt{2}$.

Poznámka 2. *Mriežkový graf patrí medzi riedke grafy, pretože má veľmi malý počet hrán (lineárny od počtu vrcholov).*



Obr. 1.2: Mriežkový graf 2x2



Obr. 1.3: Mriežkový graf bez označenia vrcholov a dĺžok hrán

Zadefinujme si teraz mriežkový graf formálne.

Definícia 6. *Mriežkový graf rozmerov $m \times n$ je ohodnotený graf s ohodnotením w s $m \cdot n$ vrcholmi očíslovanými od $v_{1,1}$ až po $v_{m,n}$ s priamymi hranami j v tvare $\{v_{a,b}, v_{a,b+1}\}, \{v_{a,b}, v_{a+1,b}\}$, kde $w(j) = 1$ a šikmými hranami s v tvare $\{v_{a,b}, v_{a+1,b+1}\}$, $\{v_{a,b}, v_{a-1,b+1}\}$, kde $w(s) = \sqrt{2}$.*

Poznámka 3. *Mriežkový graf sa dá reprezentovať ako matica $m \times n$ nad telesom $\mathbb{Z}_2$, kde jednotky predstavujú miesta, kde sú vrcholy.*

Poznámka 4. *Konkrétny vrchol $v_{x,y}$ sa dá taktiež popísať jedným číslom $c := x \cdot n + y$, kde n je počet stĺpcov matice.*

Príklad 1. *Mriežkový graf rozmerov 2×2 s vyznačenými dĺžkami hrán vidíme na obrázku 1.2. Príklad mriežkového grafu 4×7 je na obrázku 1.3. Príklad jeho maticovej reprezentácie je na obrázku 1.4.*

$$G = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Obr. 1.4: Maticová reprezentácia mriežkového grafu

1.3 GPPC: Grid-Based Path Planning Competition

Algoritmus navrhnutý a naprogramovaný v tejto práci bol zaradený do súťaže **GPPC**, ktorá sa koná približne raz ročne.

1.3.1 Špecifiká súťaže, limity

Mriežkové grafy budú mať rozmery maximálne 2048×2048. Súťaž bude rozdelená do dvoch fáz — fázy predspracovania grafu (pre-processing) a fázy testovania. Na predspracovanie grafu bude vyhradený čas maximálne 30 minút a program si svoje dáta uloží na disk do súboru o veľkosti maximálne 50MB. Potom vo fáze testovania budé dostávať požiadavky na nájdenie najkratšej cesty. Úlohou je naimplementovať interface zobrazený na obrázku 1.5.

Funkcia *PreprocessMap* má za úlohu predspracovať mriežkový graf a vytvoriť si pomocné dátové štruktúry podľa predpísaných limitov. Jej prvým argumentom je samotný mriežkový graf predaný ako zlinearizovaná matica, ktorú bolo vidieť

```
struct xyLoc
{
    int x;
    int y;
};

void PreprocessMap(std::vector<bool> &bits, int width,
 int height, const char *filename);

void *PrepareForSearch(std::vector<bool> &bits, int width,
 int height,const char *filename);

bool GetPath(void *data, xyLoc s, xyLoc g, std::vector &path);
```

Obr. 1.5: Interface, ktorý treba naimplementovať

na obrázku 1.4. Ďalšími argumentami sú rozmery grafu, keďže z jednorozmernej reprezentácie nie sú odvoditeľné. Posledným argumentom je názov súboru, do ktorého sa pomocné dátové štruktúry uložia.

Funkcia *PrepareForSearch* slúži na načítanie dát z vytvoreného súboru. Argumenty má rovnaké, ako predošlá funkcia. Štvrtý argument dodáva názov súboru, z ktorého sa pomocné dáta načítajú. Funkcia po načítaní dát skonštruuje dátové štruktúry a vráti na nich smerník.

Tento smerník prevezme v prvom argumente funkcia *GetPath*, ktorá slúži na nájdenie najkratšej cesty. Jej ďalšími argumentami sú súradnice počiatočného a koncového bodu, pre ktoré sa bude hľadať cesta a posleným argumentom je referencia na vektor, do ktorého sa samotná cesta uloží. Funkcia vracia hodnotu typu boolean na základe toho, či už vypočítala istý úsek a chce vrátiť cestu. Zíde sa napríklad, keď treba vrátiť čo najrýchlejšie aspoň prvých k krokov cesty (viď. Kritériá, sekcia 1.3.2).

1.3.2 Kritériá súťaže, hodnotenie programov

Na algoritmus môžeme klásť rôzne požiadavky, ktoré si častokrát navzájom protirečia, preto programy posudzujeme podľa viacerých kritérií.

- Celkový čas na nájdenie cesty.
- Čas na nájdenie prvých 20 tich krokov.
- Dĺžka cesty (zohľadnená suboptimalita).
- Maximálny čas vrátenia hociktorej časti cesty.

Testovací počítač má 12 GB RAM pamäti a dva 2.4 GHz Intel Xeon E5620 procesory.

2. Prehľad algoritmov

Na hľadanie najkratších ciest v grafe poznáme mnoho algoritmov, ktoré rozdeľujeme do týchto troch [4] skupín:

- Point To Point Shortest Path(P2PSP) - hľadajú najkratšiu cestu medzi dvoma zadanými bodmi.
- Single Source Shortest Path(SSSP) - pre daný vrchol v hľadajú najkratšiu cestu do všetkých vrcholov grafu.
- All Pairs Shortest Path (APSP)- skúmajú najkratšiu cestu medzi všetkými dvojicami vrcholov.

Tieto problémy sú na obecných grafoch NP-ťažké. Napriek tomu na mriežkových grafoch (kde sú vzdialenosti medzi vrcholmi vždy kladné) existujú algoritmy v polynomiálnom čase.

V práci sa budeme ďalej zaoberať riešením prvého problému (Point to Point Shortest Path).

V tejto kapitole popíšeme algoritmy, ktoré sú použiteľné na grafoch s nezápornými dĺžkami hrán.

2.1 Kritériá efektivity algoritmu

Na porovnanie efektivity algoritmou slúži v teoretickej informatike odhad asymptotickej zložitosti [8]. Tento odhad je veľmi užitočný v teoretickej informatike a veľmi často algoritmus s lepšou zložitosťou je v praxi rýchlejší. Nie je to ale pravidlom a teda potrebujeme zaviesť ďalšie kritériá, ktoré presnejšie popíšu a porovnajú správanie algoritmov v praxi. Kritéria, podľa ktorých budeme porovnávať efektivitu algoritmov sú teda nasledovné:

- Asymptotická zložitosť.
- Počet navštívených vrcholov.
- Reálny čas behu algoritmu.

2.2 Dijkstrov algoritmus

Medzi základné algoritmy typu SSSP patrí Dijkstrov algoritmus [2] popísaný už v roku 1959. Miernu modifikáciu pôvodného algoritmu môžeme vidieť na Algoritme 1). Patrí medzi relaxačné algoritmy a zbehne korektne na grafoch s nezápornými hranami.

Pri hľadaní cesty z vrcholu s do vrcholu t prechádzame postupne vrcholy s neklesajúcou vzdialenosťou od s, až dokým sa nedostaneme k cieľovému vrcholu t.

Vrchol môže byť v jednom z troch stavov: NENAVŠTÍVENÝ, OTVORENÝ a ZATVORENÝ. Nenavštívený bude vrchol, do ktorého sme ešte ani nezačali

hľadať najkratšiu cestu. Vrchol je otvorený, keď sme našli najkratšiu cestu k nejakému jeho susedovi a vrchol je uzavretý, pokiaľ sme už k nemu našli najkratšiu cestu. V algoritme budeme používať minimovú haldu, ktorá vracia vrcholy s najmenšou vzdialenosťou. Vrchol sa po vložení do haldy automaticky otvára.

Na začiatku sú všetky vrcholy v stave NENAVŠTÍVENÝ a vložíme do haldy počiatočný vrchol. Postupne z haldy vyberáme vrcholy a po vybratí ich uzavrieme. Po vybraní otvoreného vrcholu prejdeme všetkých jeho neuzavretých susedov a pokiaľ sme k nim našli cestu kratšiu, ako bola dosiaľ nájdená, tak ich vložíme do haldy.

Algoritmus 1 Dijkstra: zisti vzdialenosť najkratšej cesty z vrcholu s do všetkých dostupných vrcholov

Vstup: graf G
Výstup: dĺžková funkcia d obsahujúca najkratšie cesty z vrcholu s do vrcholov grafu

```
1: d(∗) ← ∞
2: stav(∗) ← NENAVŠTÍVENÝ
3: // pridám počiatok
4: d(s) ← 0
5: stav(s) ← OTVORENÝ
6: Heap H
7: Insert(H, s)
8: while H not empty do
9:    // vyberieme v — najbližší otvorený vrchol
10:   v ← ExtractMin(H)
11:   while stav(v) ≠ OTVORENÝ do
12:      v ← ExtractMin(H)
13:   end while
14:   stav(v) ← UZAVRETÝ
15:   // zrelaxujeme vrchol v
16:   for all e, e = (v, u) do
17:      if d(u) > d(v) + l(v, u) then
18:         Insert(H, v)
19:         stav(u) ← OTVORENÝ
20:         d(u) ← d(v) + l(v, u)
21:      end if
22:   end for
23: end while
```

Veta 1. *V dijkstrovom algoritme uzatvárame každý dosiahnuteľný vrchol práve raz.*

Dôkaz. Napríklad [4]. □

2.2.1 Zložitosť

Každý vrchol vložíme do haldy maximálne $deg(v)$-krát (v najhoršom prípade postupne vyberáme z haldy jeho susedov a cez každého nasledujúceho suseda

vedie kratšia cesta k vrcholu v – teda ho stále pridáme znovu). Počet všetkých vložení bude teda rádovo $\mathrm{O}\big(\sum_v deg(v)\big) = \mathrm{O}\big(m\big)$. Zo štruktúry môžme vybrať maximálne toľko prvkov, koľko sme tam vložili a teda aj volania $ExtractMin$ trvajú $\mathrm{O}\big(m\big)$.

Algoritmus zbehne v čase $\mathrm{O}\big(mT_i + mT_e\big)$, kde T_i odpovedá času na vloženie prvku a T_d odpovedá času na vybranie najmenšieho prvku.

To znamená, že zložitosť algoritmu závisí od zložitosti operácií $Insert$ a $ExtractMin$. Na riedke grafy je obecne v praxi najvýhodnejšie použiť binárnu haldu, ktorej obe operácie trvajú $\mathrm{O}\big(\log n\big)$ a celkový čas je $\mathrm{O}\big(m \log n\big)$ Prehľad štruktúr aj so zložitosťami operácií $Insert$ a $ExtractMin$ sa nachádza napr. v [4].

2.2.2 Halda na mriežkovom grafe

Nakoľko mriežkový graf je veľmi špeciálny typ grafu, vieme niektoré jeho vlastnosti využiť na to, aby sme vytvorili štruktúru, ktorá zvládne obe operácie v konštantnom čase.

Na konštrukciu tejto štruktúry (viď. [11]) budeme potrebovať nasledujúcu vetu.

Veta 2. *Pokiaľ sme v Dijkstrovom algoritme uzavreli vrchol u so vzdialenosťou $d(u)$ a najkratšia hrana v grafe má dĺžku ϵ, tak môžme taktiež uzavrieť všetky vrcholy v so vzdialenosťami $d(v) \in (d, d + \epsilon)$.*

Dôkaz. Do haldy vieme pridávať len vrcholy so vzdialenosťami aspoň $d+\epsilon$ (kratšia hrana tam už nie je), ale tie už cestu k vrcholom so vzdialenosťami $d_v \in (d, d+\epsilon)$ skrátiť nemôžu. □

Dôsledok 1. *Keď uzavrieme vrchol so vzdialenosťou d_u, môžme uzavrieť aj vrcholy so vzdialenosťami menšími, ako $d_u + \epsilon$ pričom poradie je nezávislé od skutočnej vzdialenosti vrcholov.*

Príklad 2. *Dĺžka ϵ najkratšej hrany v mriežkovom grafe je 1. Je to dĺžka akejkoľvek vodorovnej, alebo zvislej hrany. Keď teda uzavrieme vrchol so vzdialenosťou $d(u)$, môžme uzavrieť aj vrcholy so vzdialenosťami menšími, ako je $d(u) + 1$ a to v ľubovoľnom poradí.*

Tieto skutočnosti vieme výborne využiť pri konštrukcii štruktúry zvanej *priehradková halda*. Tá, využijúc vyššie uvedenú vetu, uzatvára a pridáva vrcholy bez porušenia akejkoľvek konzistencie behu algoritmu.

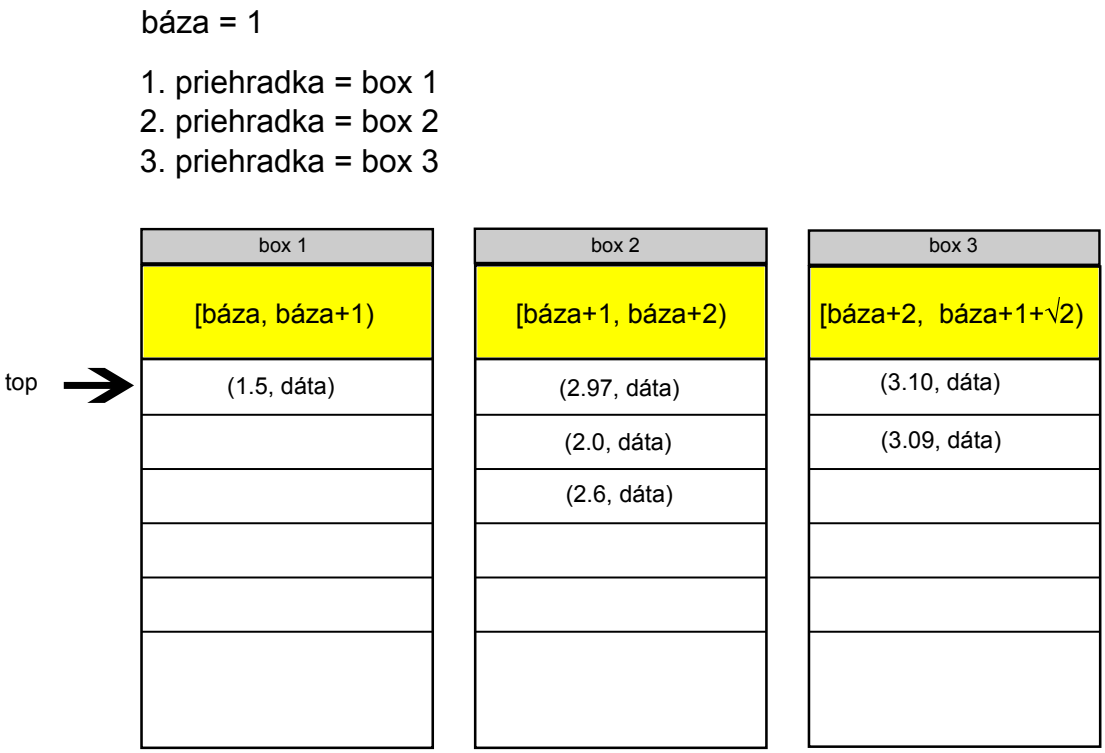
2.2.3 Popis haldy

Najprv popíšeme fungovanie haldy a graficky znázorníme jej operácie. Neskôr dokážeme, že keď túto haldu použijeme v Dijkstrovom algoritme, tak nám bude vracať korektné výsledky.

Majme haldu s tromi priehradkami (nazvime ju $BucketHeap$), pričom rozsah jednej priehradky je ostro menší ako 1. Prvá priehradka uchováva prvky s rozsahom vzdialeností $[b, b+1)$, druhá $[b+1, b+2)$ a tretia $[b+2, b+1+\sqrt{2})$ pre danú bázu b. Pre jednoduchšiu implementáciu $b \in \mathbb{N}$. Operácia $push((dist, data))$ vloží

do haldy prvok so vzdialenosťou *dist* s pomocnými dátami *data*. Operácia *pop*() vracia ľubovoľný element z prvej priehradky. Pre jednoduchšiu implementáciu budeme mať na začiatku smerník na prvý prvok prvej priehradky a po vyhodení najmenšieho prvku tento smerník jednoducho inkrementujeme, kým to bude možné. Keď už v prvej priehradke nezostane žiaden prvok a zavoláme operáciu *pop*(), vykoná sa nasledujúca operácia: druhú priehradku presunieme na miesto prvej, tretiu na miesto druhej a prvú dáme namiesto tretej.

Ilustrujme si to na obrázkových príkladoch. Príklad troj–priehradkovej haldy vidíme na obrázku 2.1. Halda uchováva premennú *baza* definujúcu bázu, od ktorej sa rozsahy priehradok odvíjajú. Okrem nej, uchováva tri smerníky na tri po sebe idúce priehradky a smerník na vrchol haldy, zvaný *top*.

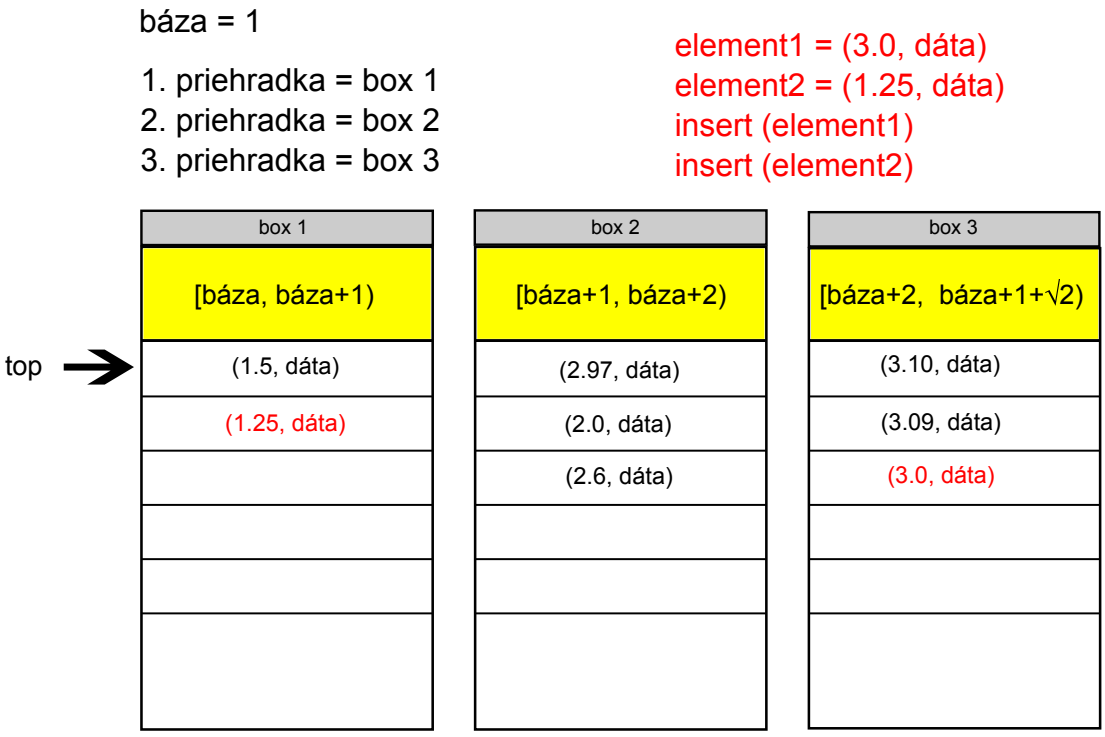


Obr. 2.1: Priehradková štruktúra s niekoľkými prvkami.

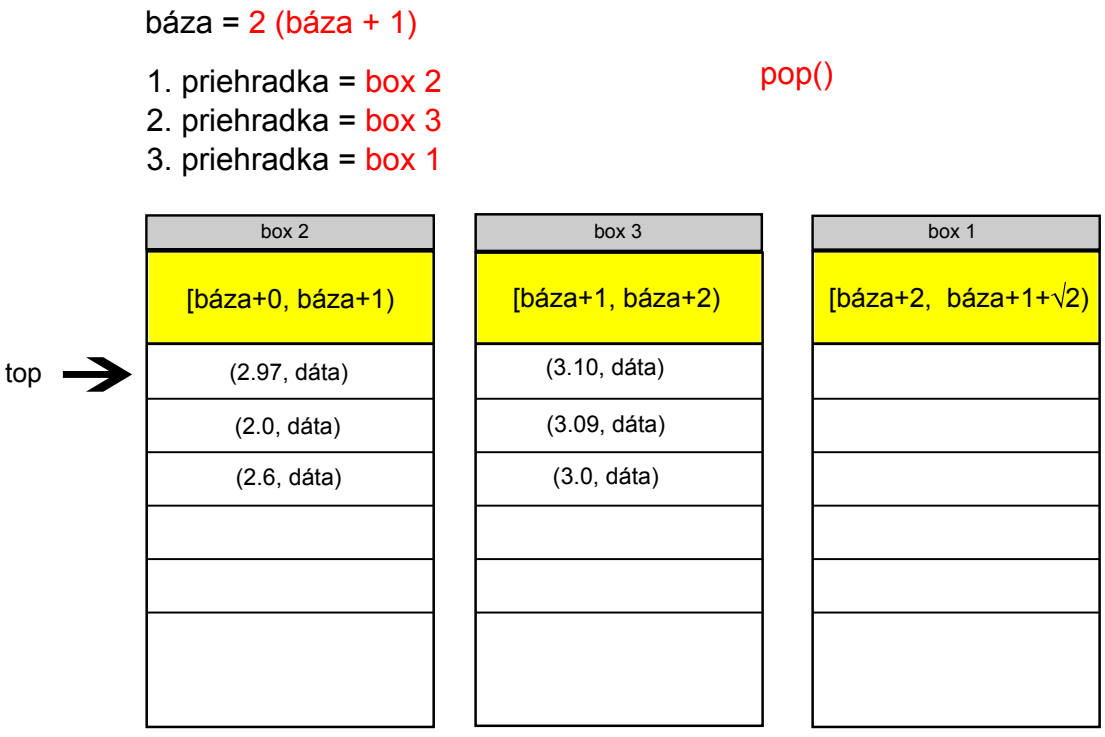
Pridanie dvoch prvkov je znázornené na obrázku 2.2. Priehradka, do ktorej má byť prvok s danou vzdialenosťou vložený sa vypočíta podľa vzorca: $\lfloor vzdialenostPrvku - baza + 1 \rfloor$.

Zmazanie prvku vidíme na obrázku 2.3. Celé zmazanie spočíva v inkrementácii ukazovateľa na vrchol haldy.

Pokiaľ sa v priehradke nachádza jediný prvok a ten chceme vybrať, tak nám inkrementácia premennej *top* v priehradke nepostačí. Musíme prehodiť priehradky. Druhú priehradku presunúť na miesto prvej, tretiu na miesto druhej a prvú umiestniť nakoniec. Viď obrázok 2.4. Nakoniec musíme zvýšit bázickú vzdialenosť. Zmena poradia týchto priehradok sa samozrejme uskutočňuje cez prehodenie smerníkov. Keďže máme konštatný počet priehradok, tak aj táto operácia trvá konštantný čas.

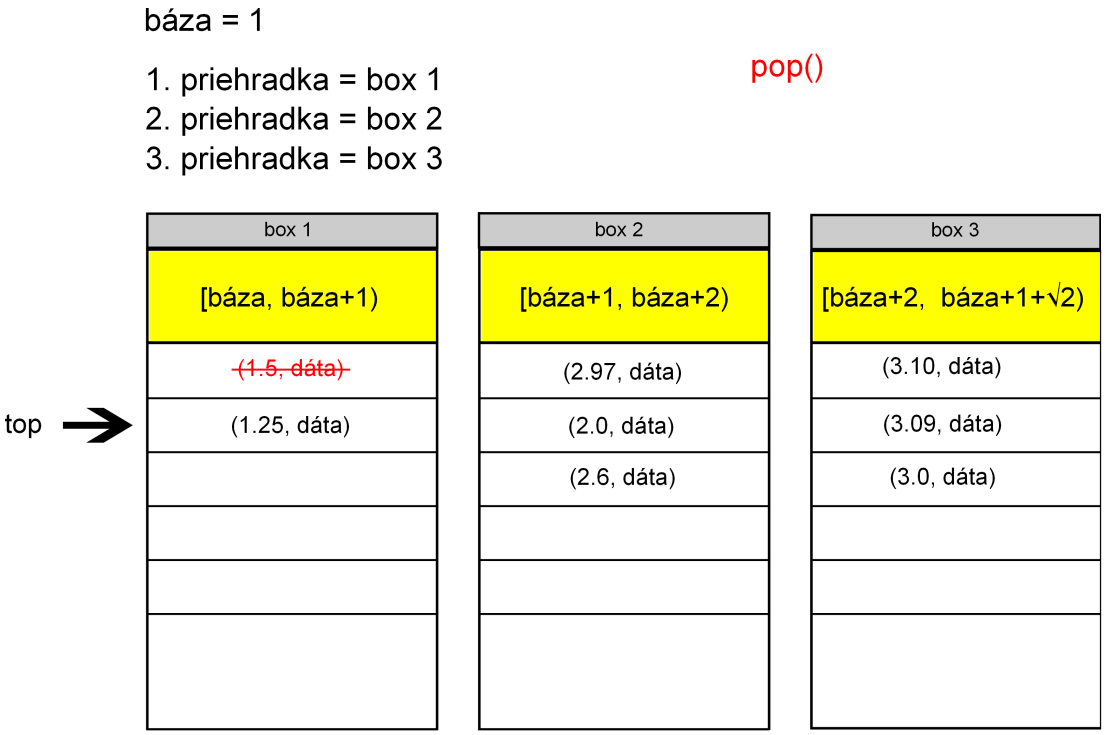


Obr. 2.2: Priehradková štruktúra po vložení dvoch vrcholov so vzdialenosťami 1.25 a 3.0. Prvky sa vkladajú stále na koniec priehradok.



Obr. 2.4: Zmazanie posledného prvku prvej priehradky vedie k zmene poradia priehradok

Veta 3 (korektnosť priehradkovej štruktúry)**.** *Dijkstrov algoritmus používajúci haldu BucketHeap vráti korektné najkratšie vzdialenosti do vrcholov grafu.*



Obr. 2.3: Vybranie prvku — Inkrementujeme smerník na vrchol priehradky.

Dôkaz. Rozsah každej priehradky je ostro menší ako 1. To znamená, že podľa vety 2 a príkladu 2 operácia *pop*() vracia prvky v poradí, ktoré nepokazí chod algoritmu.

Treba ešte dokázať, že tri priehradky postačujú. To je zrejmé, pretože keď vyberieme vrchol z prvej priehradky, tak jeho vzdialenosť je v rozsahu $[b, b + 1)$. Keď prechádzame jeho susedné vrcholy, tak najdlhšia hrana je $\sqrt{2}$ a teda vzdialenosť k najvzdialenejšiemu susednému vrcholu je ostro menšia, ako $b + 1 + \sqrt{2}$, čo sa zmestí do intervalu poslednej priehradky. □

2.3 A*

Ďalší algoritmus, ktorým sa budeme zaoberať, je algoritmus A* [5] prvykrát popísaný v roku 1968.

Tento algoritmus vychádza z Dijkstrovho algoritmu a je mu veľmi podobný. Hlavný rozdiel medzi týmito algoritmami je, že kým Dijkstrov algoritmus vyberá z haldy vrcholy s neklesajúcou vzdialenosťou $d(v)$ od počiatku, tak algoritmus A* vyberá prvky s neklesajúcou vzdialenosťou $f(v) := d(s, v) + h(v, t)$, kde $h(v, t)$ značí heuristickú funkciu, ktorá je dolným odhadom vzdialenosti od vrcholu v do cieľa t. Obrátene, Dijkstrov algoritmus si vieme predstaviť ako algoritmus A*, kde $\forall v \in G : h(v, t) = 0$.

Použitá heuristická funkcia má dopad na počet prehľadaných vrcholov a teda do zásadnej miery ovplyvňuje výkon algoritmu.

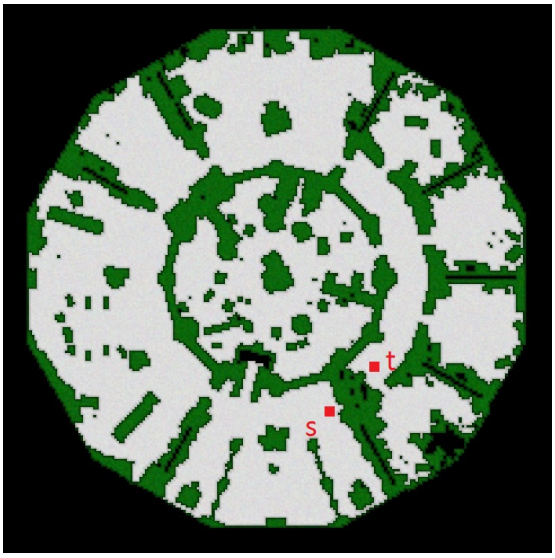
2.3.1 Heuristická funkcia

Heuristická funkcia nemôže byť ľubovoľná. Funkcia musí predstavovať tzv. *prípustný potenciál*. Podrobnejší popis sa nachádza napr. v [4] [6] [7]. Ďalej sa budeme venovať len funkciám, ktoré túto podmienku splňujú.

Najčastejšie heuristické funkcie sú tieto:

- Euklidovská vzdialenosť.
- Trojuholníková nerovnosť, tzv. *landmarks*.

Euklidovská vzdialenosť je najjednoduchšie implementovateľná heuristická funkcia. Na jednoduchých grafoch s malým počtom prekážok vracia relatívne dobré výsledky. Problém nastáva na grafoch, kde začiatok a koniec cesty sú geometricky blízko seba, hoci ich skutočná vzdialenosť je veľká. Príklad vidíme na obrázku 2.5.

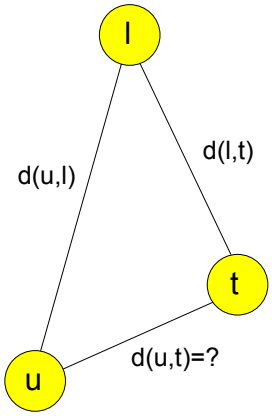


Obr. 2.5: Mapa, na ktorej euklidovská heuristika zlyhá.

Landmarks a trojuholníková nerovnosť Nevýhoda použitia euklidovskej heuristickej funkcie je na obecných mriežkových grafoch zjavná. To motivovalo vymyslieť heuristiku, ktorá lepšie odráža vzdialenosti v grafe.

Jednou z týchto heuristík je počítanie dolného odhadu pomocou tzv. *landmarks*. Landmarks sú vybrané vrcholy v grafe, z ktorých je následne prepočítaná najkratšia vzdialenosť do všetkých ostatných vrcholov grafu. Keďže jeden prechod grafu vieme Dijkstrovým algoritmom s priehradkami vykonať za lineárny čas, predpočítanie k landmarkov trvá $\mathrm{O}(kn)$, kde n značí počet vrcholov grafu.

Na výpočet dolného odhadu vzdialenosti z vrcholu u do vrcholu t cez landmark l využijeme trojuholníkovú nerovnosť. Vďaka nej platí $d(u,l) + d(u,t) > d(l,t)$ a taktiež $d(l,t) + d(u,t) > d(u,l)$. Z toho vyplýva $d(u,t) > \max\{d(l,t) - d(l,u), d(l,u) - d(l,t)\}$, kde $d(x,y)$ označuje najkratšiu vzdialenosť z bodu x do bodu y. Pre väčšiu prehľadnosť označme dolný odhad vzdialenosti $d(u,t)$ získaný výpočtom cez landmark l, ako $d_l(u,t)$ Túto skutočnosť zobrazuje obrázok 2.6.



Obr. 2.6: Hľadáme dolný odhad najkratšej cesty z u do t, použijúc landmark l

Pri použití viacerých landmarkov(nazvime ich $l_1, \ldots, l_k$) vieme zistiť kvalitnejší dolný odhad vzdialenosti. ktorý získame ako maximum dolných odhadov cez tieto landmarky. Konkrétne $d(u,t) > \max_{i \in 1,\ldots,k}\{d_{l_i}(u,t)\}$.

Možnosti voľby landmarkov Pri voľbe landmarkov zvažujeme dva faktory: počet a rozmiestnenie.

Príklad 3 (na počte záleží)**.** *Pokiaľ zvolíme málo landmarkov, tak dolný odhad nebude presný. Pokiaľ ich zvolíme priveľa, tak prepočet vzdialenosti cez každý landmark pre každý vrchol zaberie veľa času.*

Príklad 4 (na rozmiestnení záleží)**.** *Pokiaľ zvolíme všetky landmarky hneď pri sebe, tak heuristika nám nebude vracať dostatočne presné dolné odhady na vzdialenosť vrcholov, ktoré sú ďaleko od landmarkov.*

Poznámka 5. *Experimentálne sme zistili, že algoritmus funguje najrýchlejšie, ked používame 6 landmarkov. Aby landmarky dávali kvalitné dolné odhady (a teda aby nanastal problém popísaný v poznámke 4), získali sme ich pomocou tohto algoritmu: náhodne sme zvolili 3 vrcholy grafu a prehlásili sme ich za landmarky. Potom sme v týchto vrcholoch spustili Dijkstrov algoritmus, ktorým sme našli 3 najvzdialejšie body od týchto landmarkov. Tie sme tiež prehlásili za landmarky.*

Časť II

Implementácia

3. Súťažný algoritmus

Súťažný algoritmus bude využívať poznatky popísané v predošlej kapitole. Navyše zavedieme koncept tzv. *mriežkového grafu bez prekážok*, ktorý umožní mierne zrýchliť výkon algoritmu v mnohých prípadoch, väčšinou však pri trasách, kde počiatočný a koncový bod ležia relatívne „blízko seba“.

3.1 Mriežkový graf bez prekážok

Nie všetky najkratšie cesty musia obchádzať veľa prekážok. V mnohých prípadoch neleží medzi počiatočným a koncovým bodom žiadna prekážka, a teda cesty sú veľmi priamočiare. To sa pokúsime využiť na zlepšenie výkonu algoritmu. Pre ľahšie vyjadrovanie si zaveďme definíciu *mriežkového grafu bez prekážok*.

Definícia 7. *Mriežkový graf je bez prekážok, pokiaľ medzi každými dvoma susednými vrcholmi existuje hrana.*

Kvôli lepšej prehľadnosti a stručnosti budeme mriežkový graf bez prekážok nazývať aj *obdĺžnik*. Intuitívne, kvôli jeho vizuálnej predstave. Jeho obsahom bude počet vrcholov patriacich do tohto obdĺžniku.

Majme mriežkový graf bez prekážok a hľadajme najkratšiu cestu medzi bodmi $s = (x_s, y_s), t = (x_t, y_t)$. V tomto prípade vieme nájsť najkratšiu cestu veľmi jednoducho.

Algoritmus 2 Nájdi najkratšiu cestu medzi dvoma bodmi s a t na mriežkovom grafe bez prekážok

Vstup: $s = (x_s, y_s), t = (x_t, y_t)$
Výstup: *path*

```
1: path.append((x_s, y_s)) {pridám počiatok}
2: while x_s ≠ x_t ∨ y_s ≠ y_t do
3:    if x_s < x_t then
4:       x_s ← x_s + 1
5:    else if x_s > x_t then
6:       x_s ← x_s − 1
7:    end if
8:    if y_s < y_t then
9:       y_s ← y_s + 1
10:   else if y_s > y_t then
11:      y_s ← y_s − 1
12:   end if
13:   path.append((x_s, y_s))
14: end while
```

Jednoducho povedané: keď sa počiatočný a koncový bod líšia v jednej súradnici, tak sa posúvame priamočiaro, keď sa líšia v oboch, tak sa posúvame šikmo.

Pokiaľ si zadefinujeme $dx := |x_t - x_s|$ a $dy := |y_t - y_s|$, tak počet vrcholov, ktorými cesta prechádza vieme zhora odhadnúť, ako $\max(dx, dy)$. Jej vzdialenosť

vieme zistiť v čase $\mathrm{O}\big(\max(dx, dy)\big)$ Na zistenie vzdialenosti v každom kroku nám postačí konštantná pamäť.

Pokiaľ sa počiatočný a koncový bod cesty nachádza v jednom obdĺžniku, tak vieme pomocou tohto algoritmu veľmi rýchlo nájsť najkratšiu cestu. Jediným problémom ostalo rozdeliť mriežkový graf na tieto obdĺžniky.

3.2 Hľadáme obdĺžniky

Našou snahou je dekomponovať mriežkový graf do obdĺžnikov tak, aby bola čo najväčšia pravdepodobnosť toho, že dva náhodne zvolené body (začiatok a koniec) ležia v jednom obdĺžniku. Čiže aby sa dalo toto zrýchlené hľadanie najkratšej cesty použiť v najväčšom možnom počte prípadov.

3.2.1 Proporcie obdĺžnikov

Dôležitou otázkou je, na akých vlastnostiach obdĺžnikov záleží. Uvažujme nasledujúci motivačný príklad.

Príklad 5. *Majme na mriežkovom grafe nájdené dva obdĺžniky, ktoré dovedna pokrývajú 10 vrcholov. Predstavme si tieto dva prípady. V prvom prípade prvý pokrýva 9 vrcholov, druhý 1. V druhom prípade obdĺžniky pokrývajú 6 vrcholov a 4 vrcholy. Chceme maximalizovať pravdepodobnosť toho, aby pri voľbe dvoch náhodných bodov boli oba body v rovnakom obdĺžniku.*

Úlohu vieme zobecniť na klasickú pravdepodobnostno-optimalizačnú úlohu.

Príklad 6. *Máme k ekvivalenčných tried na množine s n prvkami. Ako zvoliť ekvivalenčné triedy tak, aby pri voľbe dvoch náhodných prvkov bola pravdepodobnosť toho, že oba prvky budú v tej istej ekvivalenčnej triede čo najvyššia?*

Poznámka 6. *Ekvivalenčnú triedu predstavuje obdĺžnik a množinu predstavuje množina vrcholov grafu. Alternatívne sa môžeme na úlohu pozerať ako na problém farbenia n guličiek pomocou k farieb.*

Zapíšme túto úlohu formálne. Majme n-prvkovú množinu

$$Prv = \{x_1, \ldots, x_n\}$$

k-prvkovú množinu ekvivalenčných tried

$$Ek = \{ek_1, \ldots, ek_k\}.$$

veľkosť triedy $\|ek_i\|$ označme k_i a zaveďme funkciu $f\colon Prv \to Ek$ ktorá roztriedi prvky do ekvivalenčných tried.

Označme výberový priestor

$$\Omega = \{(x_a, x_b) | x_a, x_b \in Prv, a \neq b\}.$$

Udalosťou A_i nazveme jav, v ktorom oba prvky patria do tej istej ekvivalenčnej triedy ek_i, teda

$$A_i = \{(x_a, x_b) | x_a, x_b \in Prv, a \neq b, f(x_a) = f(x_b) = ek_i\}.$$

Jav

$$A = \bigcup_{i=1}^{k} A_i$$

teda nastáva práve vtedy, keď oba vybrané prvky patria do rovnakej triedy.

Úlohou je navrhnúť funkciu f tak, aby pravdepodobnosť $P[A]$ bola čo najvyššia. Keďže udalosti A_i sú nezlučiteľné, môžme písať

$$P[A] = P[\bigcup_{i \in Ek} A_i] = \sum_{i \in Ek} P[A_i].$$

Ak si pravdepodobnosť každého javu rozpíšeme, dostaneme

$$\sum_{i \in Ek} P[A_i] = \sum_{i=1}^{k} \frac{\binom{k_i}{2}}{\binom{|Prv|}{2}}.$$

Keďže chceme nejak rozvrhnúť prvky v triedach ek_i, a menovateľ je konštanta, môžme ho vynechať.

Maximalizujeme hodnotu výrazu

$$\sum_{i=1}^{k} \binom{k_i}{2} = \sum_{i=1}^{k} \frac{k_i!}{(k_i-2)!2!} = \sum_{i=1}^{k} \frac{k_i(k_i-1)}{2}.$$

Po vyškrtnutí konštanty a roznásobení sme dostali nasledujúcu optimalizačnú úlohu: maximalizovať

$$\sum_{i=1}^{k} k_i^2 - k_i$$

za podmienok

$$\sum_{k=1}^{k} k_i = n, k_i \in \mathbb{N}_0.$$

Sumu si vieme rozpísať ako

$$\sum_{i=1}^{k} k_i^2 - k_i = \sum_{i=1}^{k} k_i^2 + \sum_{i=1}^{k} -k_i.$$

Druhá suma sa nasčíta na $-n$, čo je konštanta, takže nám stačí maximalizovať

$$\sum_{i=1}^{k} k_i^2.$$

Teraz nám už len zostáva použiť nerovnosť

$$(a+b)^2 \geq a^2 + b^2,$$

ktorá platí pre $a, b \geq 0$, z ktorej jasne vyplýva, že potrebujeme spraviť ľubovoľné k_i čo najväčšie a teda aj ekvivalenčné triedy musia byť čo najväčšie.

Potrebujeme teda nájsť obdĺžniky s najväčším možným obsahom. V programe tento problém rieši trieda Colorizator. Tá v grafe nájde najväčší obdĺžnik a „ofarbí“ ho jednou farbou. Na zvyšnom neofarbenom grafe znovu hľadá najväčší obdĺžnik a ofarbí ho inou farbou, atď. Farbenie skončí, keď obsah najväčsieho obdĺžnika je menší ako 500.

3.2.2 Nájdenie najväčšej jednotkovej podmatice

Ako sme si v úvode povedali, mriežkovú mapu vieme reprezentovať ako maticu, čiže problém môžeme ekvivalentne zapísať ako problém hľadania najväčšej jednotkovej podmatice. Tento problém má riešenie v čase lineárnom od počtu vrcholov, takže nájdenie k najväčších jednotkových matíc trvá $\mathrm{O}\big(k \cdot n\big)$, kde n je počet vrcholov matice.

Algoritmus je popísaný v blogposte od A. Agrawala [13]. Slovný popis algoritmu 3: V prvom prechode maticou si u každého vrcholu zapamätáme počet jedničiek naľavo od neho, vrátane daného vrcholu. Tento počet jedničiek môžme nazvať aj výškou riadku vzhľadom k danému vrcholu. Výšku riadku vzhľadom k p označujeme v programe $nalavoOdPrvku(p)$. Tento prechod trvá lineárny čas.

V druhom prechode algoritmus prechádza zaradom všetky stĺpce zľava doprava. V prechádzanom stĺpci prejde každý vrchol v a zisťuje počet vrcholov nad (uloží do $DlzkaSekvencieNahor$) a pod ($DlzkaSekvencieNadol$) vrcholom v, ktoré majú výšku riadku aspoň $nalavoOdPrvku(v)$.

Obsah najväčšej podmatice vzhľadom k v teda vypočítame ako $nalavoOdPrvku(v) \cdot (sequenceSizeUp + 1 + sequenceSizeDown)$.

Algoritmus 3 Nájdenie najväčšej jednotkovej podmatice v matici *mxn*

Vstup: matica M rozmerov $m \times n$ nad telesom $\mathbb{Z}_2$

Výstup: polia DlzkaSekvencieNadol a DlzkaSekvencieNahor, z ktorých vieme vypočítať veľkosť podmatice

1: **for all** prvok p, $p \in M$ **do**
2: **if** $p = 0$ **then**
3: nalavoOdPrvku(p) $\leftarrow 0$
4: **else**
5: nalavoOdPrvku(p) $\leftarrow$ najdlhšia súvislá postupnosť jednotiek končiaca prvkom p
6: **end if**
7: **end for**
8: **for** stĺpec s, $s \in M$ **do**
9: Vytvor nový zásobník na dvojice čísel (riadok, nalavoOdPrvku)
10: // prejdem stĺpec zhora nadol
11: **for** riadok r, $r \in M$, prechádzaný zhora nadol **do**
12: $p \leftarrow (s, r)$
13: **while** Zásobník je neprázdny **do**
14: vyberiem prvok *top* z vrcholu zásobníka
15: **if** *top*.nalavoOdPrvku > nalavoOdPrvku(p) **then**
16: prvokZoZasobnika $\leftarrow$ (*top*.r, s)
17: DlzkaSekvencieNadol(prvokZoZasobnika) = r - prvokZoZasobnika.r -1
18: **else**
19: Vložím prvok *prvokZoZasobnika* do zásobníka
20: break
21: **end if**
22: **end while**
23: Do zásobníka vložím dvojicu (r, nalavoOdPrvku(p))
24: **end for**
25: // prešli sme stĺpec, teda skončíme
26: **while** Zásobník je neprázdny **do**
27: vyberiem prvok *top* z vrcholu zásobníka
28: prvokZoZasobnika $\leftarrow$ (*top*.r, s)
29: DlzkaSekvencieNadol(prvokZoZasobnika) = m - prvokZoZasobnika.r - 1
30: **end while**
31: **end for**

Algoritmus 4 pokračovanie

```
// prejdem stĺpec zdola nahor
for riadok r, r ∈ M, prechádzaný zdola nahor do
  p ← (s, r)
  while Zásobník je neprázdny do
    vyberiem prvok top z vrcholu zásobníka
    if top.nalavoOdPrvku > nalavoOdPrvku(p) then
      prvokZoZasobnika ← (top.r, s)
      DlzkaSekvencieNahor(prvokZoZasobnika) = prvokZoZasobnika.r - r - 1
    else
      Vložím prvok prvokZoZasobnika do zásobníka
      break
    end if
  end while
  Do zásobníka vložím dvojicu (r, nalavoOdPrvku(p))
end for
// prešli sme stĺpec, teda skončíme
while Zásobník je neprázdny do
  vyberiem prvok top z vrcholu zásobníka
  prvokZoZasobnika ← (top.r, s)
  DlzkaSekvencieNahor(prvokZoZasobnika) = prvokZoZasobnika.r
end while
```

4. Testovanie a výsledky

4.1 Kritériá a popis testovania

4.1.1 Vstupné dáta

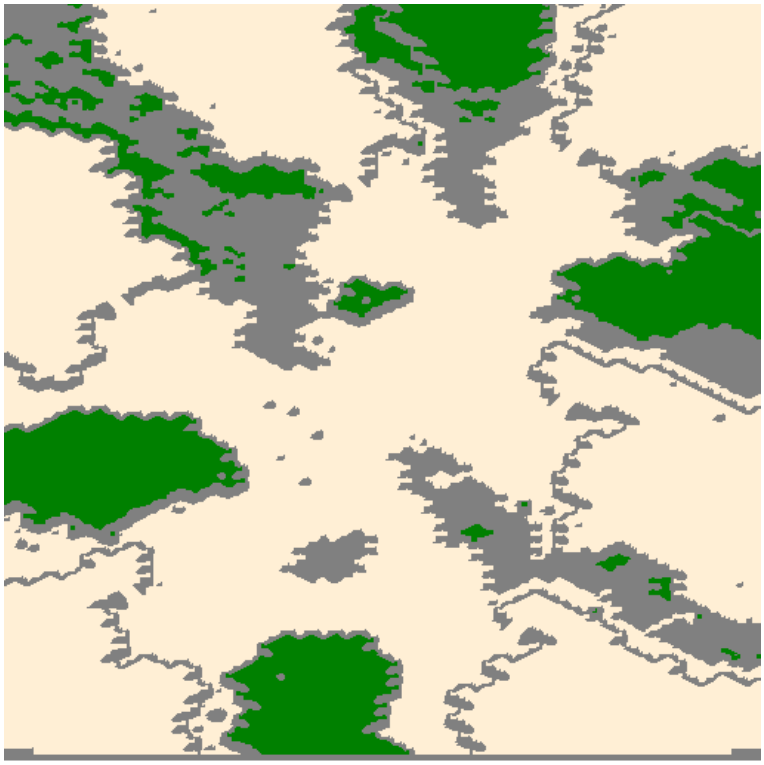
Častým problémom pri vzájomnom porovnávaní algoritmov je nájsť testovaciu vzorku, ktorá by otestovala beh algoritmu na širokej škále grafov. Ako bolo v úvode spomenuté, súťaž *Grid-Based Path Planning Competition* poskytuje množstvo máp rôznych typov a rozmerov, na ktorých sa algoritmy dajú testovať. Formát máp popisuje článok [10].

Programy boli testované na troch mapách. Ich grafická reprezentácia bola vytvorená pomocou programu T-map (popis programu a významu farieb sa nachádza v prílohe A) a môžme ju vidieť na obrázkoch Obr. 4.1, Obr. 4.2 a Obr. 4.3. Pripomeňme si, že žltou farbou je označená priechodná oblasť; ostatné oblasti sú nepriechodné.

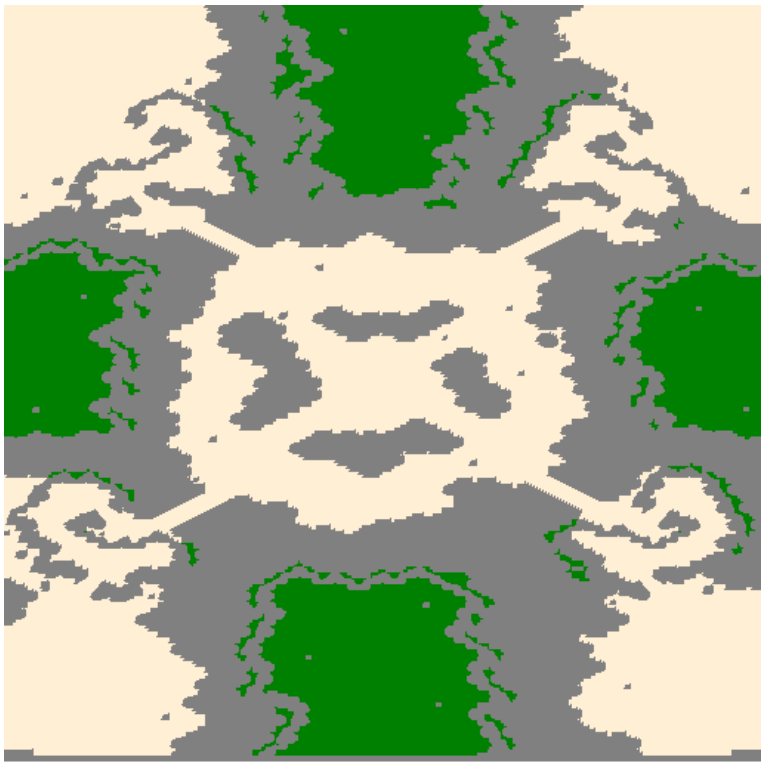
Každá mapa bola testovaná oproti dvom množinám vstupov. Každá z týchto množín obsahuje približne 200 dopytov na najkratšiu vzdialenosť medzi dvojicou zadaných bodov. Jedna z tých dvoch množín obsahuje dvojice bodov, ktoré ležia relatívne „blízko“ seba, druhá množina obsahuje dvojice bodov, medzi ktorými je vzdialenosť väčšia.

Konkrétne máme tieto 4 vstupné súbory:

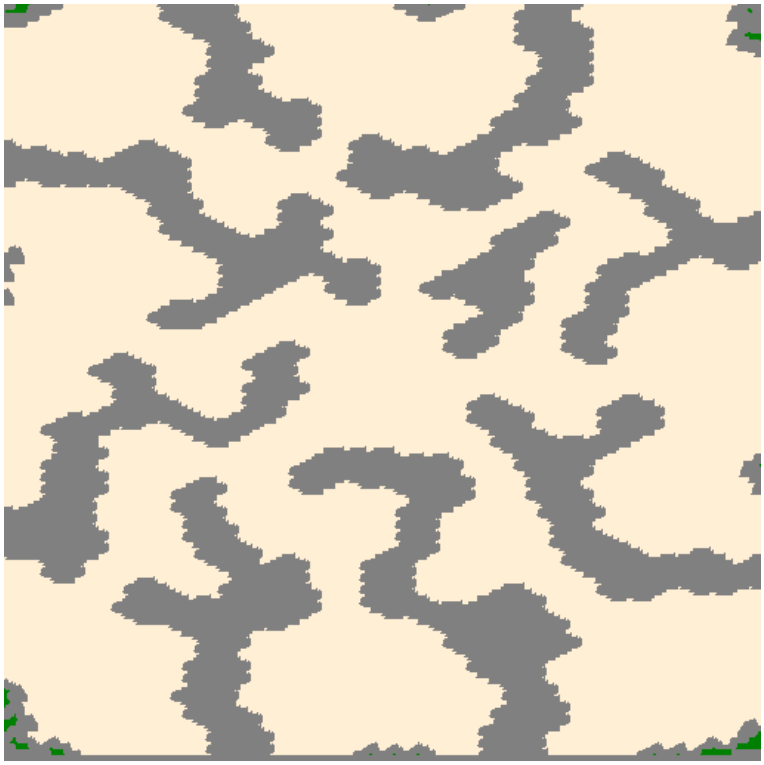
- AS — vstupný súbor k mape Aftershock obsahujúci „blízke“ body.
- AL — vstupný súbor k mape Aftershock obsahujúci vzdialenejšie body.
- BS — vstupný súbor k mape Brushfire obsahujúci „blízke“ body.
- BL — vstupný súbor k mape Brushfire obsahujúci vzdialenejšie body.
- GS — vstupný súbor k mape BigGameHunters obsahujúci „blízke“ body.
- GL — vstupný súbor k mape BigGameHunters obsahujúci vzdialenejšie body.



Obr. 4.1: Graf „Aftershock“



Obr. 4.2: Graf „Brushfire“



Obr. 4.3: Graf „BigGameHunters“

4.1.2 Testovacie kritériá

Algoritmy testujeme podľa nasledovných kritérií:

- Rýchlosť nájdenia cesty.
- Dĺžka trasy.
- Počet prehľadaných vrcholov.

4.1.3 Testované algoritmy

Testovať budeme nasledujúce algoritmy:

- Dijkstrov algoritmus nad binárnou haldou (označíme DH).
- Dijkstrov algoritmus nad priehradkovou haldou (označime DBQ).
- A* používajúci 6 landmarkov (označíme A*).
- A* používajúci 6 landmarkov a taktiež obdĺžnikovú dekompozíciu, popísanú v kapitole 3 (označíme A*+).
- Anderson – s prehľadom najrýchlejší algoritmus, ktorý vyhral súťaž (označíme And).

4.1.4 Kompilácia

Kód bude kompilovaný kompilátorom g++ s direktívou *-O3*, čo zaručuje maximálnu rýchlosť a efektivitu behu programu.

4.2 Výsledky

Tabuľka 4.1 popisuje čas v sekundách, ktorý strávil daný algoritmus hľadaním cesty medzi všetkými dvojicami zadaného vstupu. Z tabuľky môžme vypozorovať zaujímavé správanie algoritmov. Najrýchlejší je algoritmus Anderson, ktorý je zatiaľ najrýchlejším algoritmom v tejto súťaži. Ďalej vidíme, že dijkstrov algoritmus je na niektorých mapách a vstupoch niekoľkonásobne rýchlejší ako A* a na niektorých zasa pomalší. Dekompozícia mapy na obdĺžniky zrýchlila algoritmus A* len keď bola vzdialenosť medzi počiatočným a koncovým bodom dostatočne malá.

	AS	AL	BS	BL	GS	GL
DH	8.706919	8.609133	4.554813	3.803920	11.059045	7.932771
DBQ	0.506893	11.505914	0.329444	4.315670	0.467251	11.990992
A*	0.263951	7.481771	0.283890	15.599357	0.393476	9.667814
A*+	0.182731	7.502437	0.197504	15.670339	0.257229	9.694499
And	0.000543	0.002659	0.000359	0.001747	0.000673	0.002197

Tabuľka 4.1: Časy behu algoritmov nad zadanými množinami vstupov. Uvedené v sekundách

Tabuľka 4.2 uvádza súčet dĺžok najkratších ciest pre danú testovanú množinu. Vyplýva z nej, že rýchly algoritmus Anderson nie je optimálny a táto neoptimalita sa prejavuje najmä keď je vzdialenosť medzi bodmi malá. Ostatné algoritmy s výnimkou implementácie Dijkstrovho algoritmu s klasickou binárnou haldou vykazujú dobré výsledky.

	AS	AL	BS	BL	GS	GL
DH	111010	548861	78330.7	326556	82873.9	395860
DBQ	8710.17	548861	8731.65	152185	8744.62	128398
A*	8743.7	144527	8762.74	153288	8774.98	128885
A*+	8738.43	144527	8758.98	153288	8769.46	128885
And	22057.2	144854	22010	156927	27717.8	135411

Tabuľka 4.2: Dĺžky trás algoritmov. Uvedené ako súčty dĺžok hrán ciest medzi množinami dvojíc vrcholov

Posledná tabuľka 4.3 popisuje počet vrcholov grafu, ktoré algoritmus prehľadal, ked hľadal najkratšiu cestu. Táto veličina bola zisťovaná len z algoritmov autora. Z nej vyplýva, že algoritmus A* prehľadáva menej vrcholov, ako dijkstrov algoritmus. Taktiež z nej vyplýva, že dekompozícia grafu na obdĺžniky je pri vzdialených vrcholoch zbytočná.

	AS	AL	BS	BL	GS	GL
DH	20888032	26585115	14967159	15403707	25967574	22755932
DBQ	1347329	39028967	1176089	20528587	1255147	40707226
A*	350839	13854096	624561	41883515	637189	17936238
A*+	250022	13854096	465311	41883515	407731	17936238

Tabuľka 4.3: Výsledky obsahujúce počet navštívených vrcholov

Záver

Zoznam použitej literatúry

[1] L. Euler. *Solutio problematis ad geometriam situs pertinentis*. Commentarii academiae scientiarum Petropolitanae, 8:128–140, 1741.

[2] E. W. Dijkstra. *A note on two problems in connexion with graphs*. Numerische Mathematik, 1(1):269–271, 1959.

[3] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[4] M. Mareš. *Krajinou grafových algoritmů*. ITI Series, Prague 2007 [ONLINE]. [CIT. 2013-7-21]. Dostupné z `http://mj.ucw.cz/vyuka/ga/`.

[5] P. E. Hart, N. J. Nilsson, B. Raphael. Correction to a formal basis for the heuristic determination of minimum cost paths. *SIGART Bull.*, (37):28–29, December 1972.

[6] A. V. Goldberg. *Shortest Path Algorithms: Engineering Aspects*. In Proc. ESAAC '01, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[7] A. V. Goldberg, C. Harrelson. *Computing the Shortest Path: A* Search Meets Graph Theory*. In Proc. 16th ACM-SIAM Symposium on Discrete Algorithms, pages 156–165, 2005.

[8] J. Hartmanis, R. Stearns. *On the computational complexity of algorithms*. Transactions of the American Mathematical Society, vol. 117, 285–306, 1965.

[9] N. Sturtevant. *GPPC: Grid-Based Path Planning Competition* [ONLINE] 2013. [CIT. 2013-7-21]. Dostupné z: `http://movingai.com/GPPC/`.

[10] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012.

[11] A. Goldberg, C. Silverstein. *Implementations of Dijkstra's Algorithm Based on Multi-Level Buckets*. Springer Berlin Heidelberg, 1997.

[12] G. Krasner, S. Pope. A description of the Model-View-Controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[13] A. Agrawal. *Find largest sub-matrix with all 1s (not necessarily square)* [ONLINE] 2011. [CIT. 2013-7-21]. Dostupné z `http://tech-queries.blogspot.cz/2011/09/find-largest-sub-matrix-with-all-1s-not.html`

Prílohy

A. T-maps — užívateľská dokumentácia

A.1 Popis programu T-maps

Program T-maps načíta mriežkový graf vo forme znakovej matice a vykreslí ho. K danému grafu môže voliteľne načítať dátový súbor, ktorý obsahuje informácie o behu algoritmu: informáciu o navštívených vrcholoch a o najkratšej ceste. Túto maticu aj s dátami vykreslí na obrazovku. Výstup pripomína mapu používanú v počítačových hrách. Zvláda niekoľko ďalších operácií, ako je priblíženie a oddialenie mapy, posúvanie sa po mape a uloženie aktuálne prehliadaného úseku mapy.

A.2 Formát vstupu a výstupu

Presný formát vstupného súboru s mriežkovým grafom je popísaný v [10] a je zhodný s formátom, ktorý používa algoritmus. Znaková matica obsahuje rôzne znaky reprezentujúce rôzne herné prvky. Strom je reprezentovaný znakom „T“(tree), voda znakom „W“(water), priechodná oblasť(a teda znak, ktorý reprezentuje vrchol grafu) znakom „.“, prípadne „S“ (swamp).

Dátový súbor obsahuje dva riadky. Prvý riadok obsahuje medzerou oddelené čísla vrcholov, ktoré ležia na najkratšej ceste. Druhý riadok obsahuje čísla vrcholov, ktoré boli navštívené počas behu algoritmu. Vrcholy sú popísané jedným číslom tak, ako to bolo popísané v poznámke 4.

Výstup môže byť exportovaný do viacerých grafických formátov, medzi ktorými sú GIF, PNG, BMP a JPEG.

A.3 Vykreslenie znakovej matice

Znaky v matici môžu byť vykreslené dvomi spôsobmi. V takzvanom bichromatickom móde sa na vykreslenie matice použijú dve farby. Svetlejšou farbou sú vykreslené znaky reprezentujúce vrcholy matice (v hrách ide o priechodný terén), tmavšou farbou sú vykreslené znaky, ktoré nereprezentujú žiaden vrchol. Ide o stromy, vodu a podobne.

Pri plnofarebnom vykreslení sú znaky vykreslené nasledovnými farbami:

- Strom - zelená.
- Voda - modrá.
- Bažina a defaultná priechodná oblast - papájovožltá.
- Defaultná nepriechodná oblasť - sivá.

A.4 Vykreslenie dátového súboru

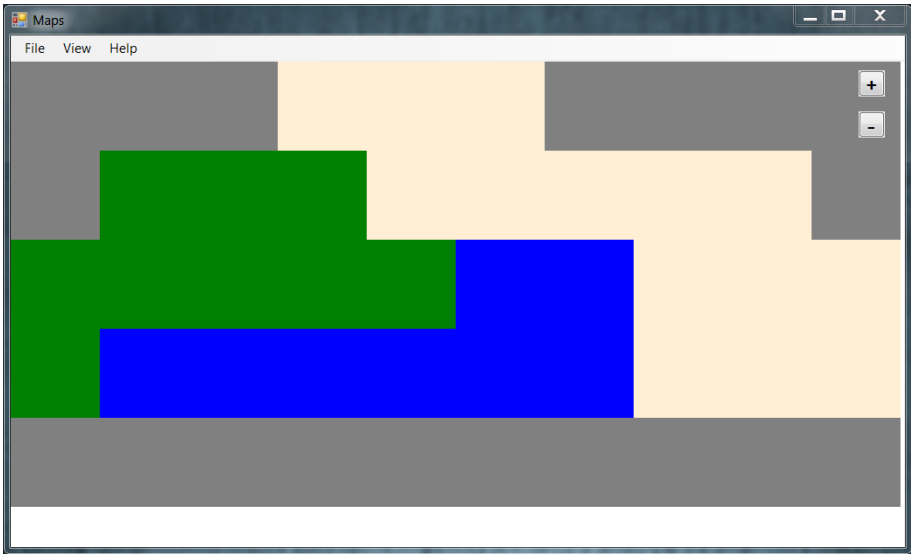
Pokiaľ bol vrchol prehľadávaný počas hľadania najkratšej cesty, jeho farba bude oranžová. Ak je dokonca súčasťou najkratšej cesty, tak bude červený.

A.5 Ukážky programu

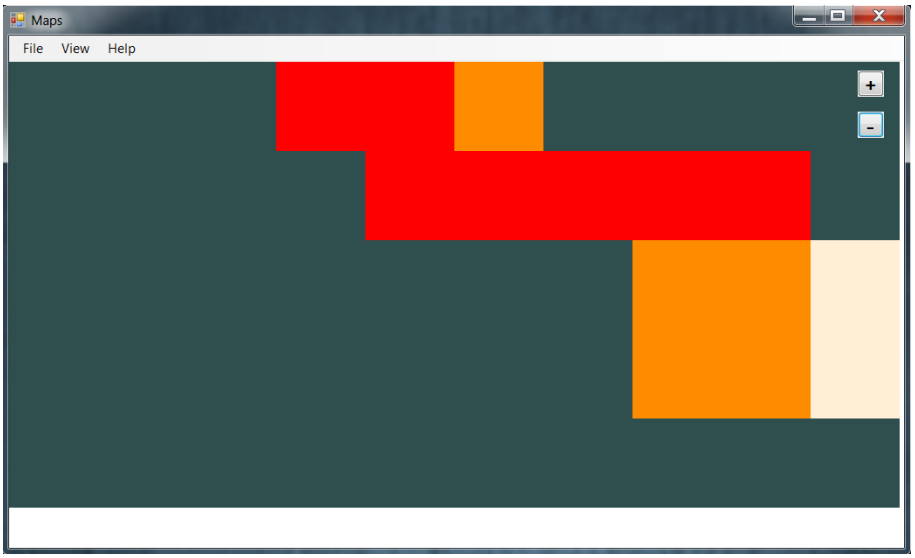
Na obrázkoch nižšie sa nachádza znaková matica veľmi malého mriežkového grafu (Obr. A.1) a ukážky z programu: screenshot programu pri načítanej znakovej matici vo farebnom móde (Obr. A.2), a screenshot programu aj s vyznačenou najkratšou cestou a prehľadanými vrcholmi v bichromatickom móde(Obr. A.3).

@	@	@	.	.	.	@	@	@	@
@	*T*	*T*	*T*	.	.	.	.	.	@
T	*T*	*T*	*T*	*T*	*W*	*W*	*S*	*S*	.
T	*W*	*W*	*W*	*W*	*W*	*W*	.	.	.
@	@	@	@	@	@	@	@	@	@

Obr. A.1: Maticová reprezentácia mriežkového grafu



Obr. A.2: Screenshot programu



Obr. A.3: Vyexportovaný výstup v bichromatickom móde aj s označenou najkratšou cestou a prehľadanými vrcholmi

B. T-maps — programátorská dokumentácia

B.1 Architektúra aplikácie

Architektúra programu *T-maps* je inšpirovaná konceptom MVC [12]. Model tvoria dve triedy: Map (Obr. B.1) a CachedBitmap (Obr. B.2). View a Controller sú spojené triede Form1.

Aplikácia bola navrhnutá s dôrazom na nízke systémové nároky aj pri operáciách, ako je nekonečný zoom. Ten je riešený tak, že v triede *CachedBitmap* je uložená bitmapa odpovedajúca výseku znakovej matice, ktorá sa v prípade potreby prepočíta. Rozmery tejto bitmapy odpovedajú dvojnásobku rozmerov zobrazovacieho okna aplikácie.

B.2 Kľúčové funkcie aplikácie

```
class Map
{
    char[][] data;
    char[][] map;
    public void Load(string filename);
    public void LoadData(string filename);
}
```

Obr. B.1: Kľúčové funkcie triedy Map. Obsahuje dvojrozmerné polia reprezentujúce znakovú maticu a taktiež dáta z dátového súboru

```
class CachedBitmap
{
    private Bitmap cachedBitmap;
    private bool isBichromatic;

    // dalsie premenne viazane na vlastnosti cachedBitmap
    // ...

    private Map map;
    public void setMap(Map m);
    public void DrawBitmapInto(Graphics g, Point TLPoint, Size
       ViewPortSize, int squareS, bool isBichrom, bool
       forcePrecomputing = false);

    private void PrecomputeBitmap(Point TLPoint, Size
       viewPortSize);
}
```

Obr. B.2: Kľúčové funkcie triedy CachedBitmap. Obsahuje v sebe referenciu na triedu Map a úsek veľkej mapy cachedBitmap. Jej hlavnou metódou je DrawBitmapInto, ktorej parametry špecifikujú úsek, ktorý sa má vykresliť. Metóda ho vykreslí pomocou grafického objektu g a v prípade nutnosti zavolá funkciu PrecomputeBitmap na predpočítanie mapy.