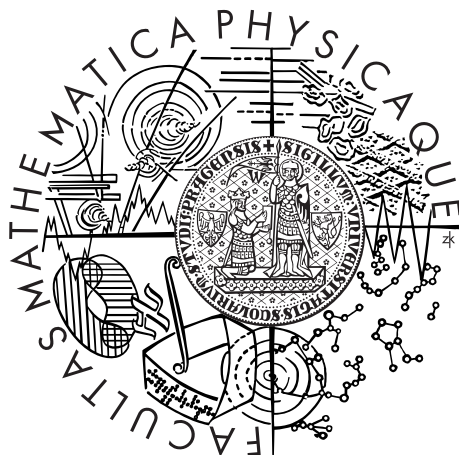


Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

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

aa vedenie práce, námietky a pripomienky a konzultácie.

Ďalej by som 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:

Klíčové slová:

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:

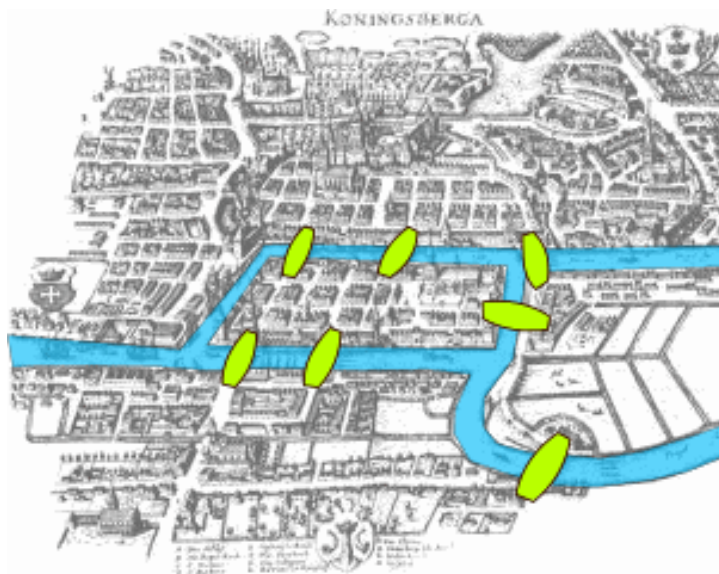
Obsah

Úvod	7
I Analýza problému	9
1 Zadanie problému a cieľové požiadavky	10
1.1 Úvodné definície a značenia	10
1.2 Mriežkový graf	11
1.3 GPPC: Grid-Based Path Planning Competition	12
1.3.1 Špecifiká súťaže, limity	12
1.3.2 Kritériá súťaže, hodnotenie programov	13
2 Prehľad algoritmov	14
2.1 Kritériá efektivity algoritmu	14
2.2 Dijkstrov algoritmus	14
2.2.1 Zložitosť	15
2.2.2 Halda na mriežkovom grafe	16
2.2.3 Popis haldy	16
2.3 A*	18
2.3.1 Heuristická funkcia	19
II Implementácia	22
3 Súťažný algoritmus	23
3.1 Mriežkový graf bez prekážok	23
3.2 Hľadáme obdĺžniky	24
3.2.1 Proporcie obdĺžnikov	24
3.2.2 Nájdenie najväčšej jednotkovej podmatice	25
4 Testovanie a výsledky	27
4.1 Kritériá a popis testovania	27
4.1.1 Vstupné dáta	27
4.1.2 Testovacie kritériá	27
4.1.3 Testované algoritmy	27
4.1.4 Kompilácia	27
4.1.5 Typy map a ciest	28
5 T-maps - Vizuálna predstava	29
5.1 Použitie	29
Záver	30
Zoznam použitej literatúry	31
Zoznam tabuliek	32

Zoznam použitých skratiek	33
Prílohy	34

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 najvýznamnejšie 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 súťaže 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 tretej kapitole navrhujeme vlastné riešenie založené na poznatkoch popísaných v druhej kapitole s pridaním vlastných vylepšení.

Vo štvrtej kapitole toto riešenie porovnáme s dosavadnými.

Časť I

Analýza problému

1. Zadanie problému a cieľové požiadavky

1.1 Úvodné definície a značenia

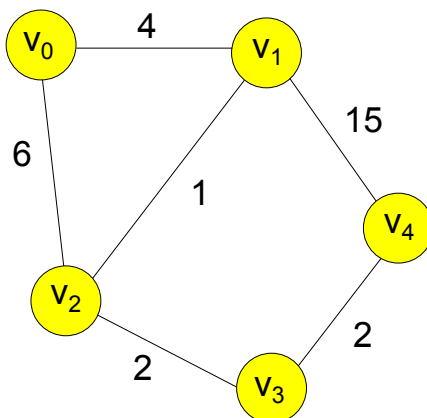
Na začiatok si zavedieme niektoré dôležité pojmy z teórie grafov. Úlohu so všetkými jej špecifikami si ozrejníme v nasledujúcich podkapitolách.

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 s spolu s reálnou funkciou (tzv. ohodnotením) $w : E(G) \rightarrow \mathbb{R}$, kde w je funkcia, ktorá každej hrane priradí reálne číslo takzvanú 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 pojmy, ako sú *cesta* a *najkratšia cesta*.

Definícia 3. Cesta P z vrcholu v_0 do vrcholu v_n v grafe G je postupnosť $P = (v_0, e_1, v_1, \dots, e_n, v_n)$, pre ktorú platí $e_i = \{v_{i-1}, v_i\}$ 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.

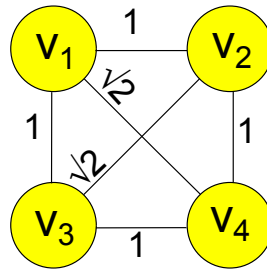
1.2 Mriežkový graf

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

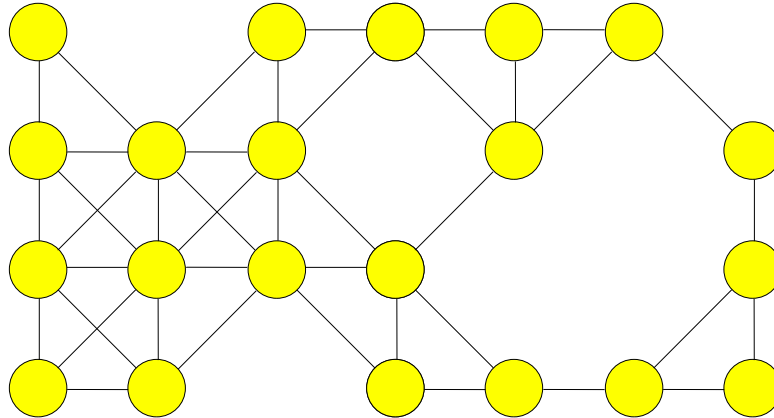
V hernej praxi predstavuje mriežkový graf mapu hracej plochy, s ktorou sa stretávame v najrôznejších hrách, ako je Warcraft, Startcraft, Dragon Age [10] a podobne.

Ide o špeciálny a dosť obmedzený typ grafu. Vizualne si ho môžeme 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 vrcholy.

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 budú dostávať požiadavky na nájdenie najkratšej cesty. Úlohou je naimplementovať interface zobrazený na obrázku 1.5. ASK?? obrazku???a keď nie obrazk, tak ake cislovanie tomu dat? cislovat znovu od jednicky?

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 na obrázku 1.4. Ďalšími argumentami sú rozmery grafu, keďže z jednorozmernej

```
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

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 kladť 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 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. TODO?? prerob

2.1 Kritériá efektivity algoritmu

Na porovnanie efektivity algoritmov slúži v teoretickej informatike odhad asymptotickej složitosti [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. Kritériá, 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 (Algoritmus 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 vybratí 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:  $d(*) \leftarrow \infty$ 
2:  $stav(*) \leftarrow \text{NENAVŠTÍVENÝ}$ 
3: // pridám počiatočok
4:  $d(s) \leftarrow 0$ 
5:  $stav(s) \leftarrow \text{OTVORENÝ}$ 
6: Heap  $H$ 
7:  $Insert(H, s)$ 
8: while  $H$  not empty do
9:   // vyberieme  $v$  — najbližší otvorený vrchol
10:   $v \leftarrow ExtractMin(H)$ 
11:  while  $stav(v) \neq \text{OTVORENÝ}$  do
12:     $v \leftarrow ExtractMin(H)$ 
13:  end while
14:   $stav(v) \leftarrow \text{UZAVRETÝ}$ 
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:       $stav(u) \leftarrow \text{OTVORENÝ}$ 
20:       $d(u) \leftarrow d(v) + l(v, u)$ 
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 (zhodou náhod 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 $O(\sum_v \deg(v)) = O(m)$. Zo štruktúry môžeme vybrať maximálne toľko prvkov, koľko sme tam vložili a teda aj volania *ExtractMin* trvajú $O(m)$.

Algoritmus zbehne v čase $O(mT_i + mT_e)$, kde T_i odpovedá času na vloženie prvku a T_e odpovedá času na vybranie najmenšieho prvku.

To znamená, že zložitosť algoritmu závisí od zložitosti operácií *Insert* a *ExtractMin*. Na riedke grafy je obecné v praxi najvýhodnejšie použiť binárnu haldu, ktorej obe operácie trvajú $O(\log n)$ a celkový čas je $O(m \log n)$. 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. ASK?? moze takto pouzivat vid???

Veta 2. *Pokiaľ sme v Dijkstrovom algoritme uzavreli vrchol u so vzdialenosťou $d(u)$ a najkratšia hrana v grafe má dĺžku ϵ , tak môžeme 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. \square

Dôsledok 1. *Keď uzavrieme vrchol so vzdialenosťou d_u , môžeme uzavrieť aj vrcholy vo 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 $d(u)$, môžeme uzavrieť aj vrcholy so vzdialenosťami menšími, ako je $d(u) + 1$ a to v ľubovoľnom poradí.*

Tieto veci vieme výborne využiť pri konštrukcii štruktúry zvanej *priehradková halda*. Tá, využívajúc 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 vracat korektné výsledky.

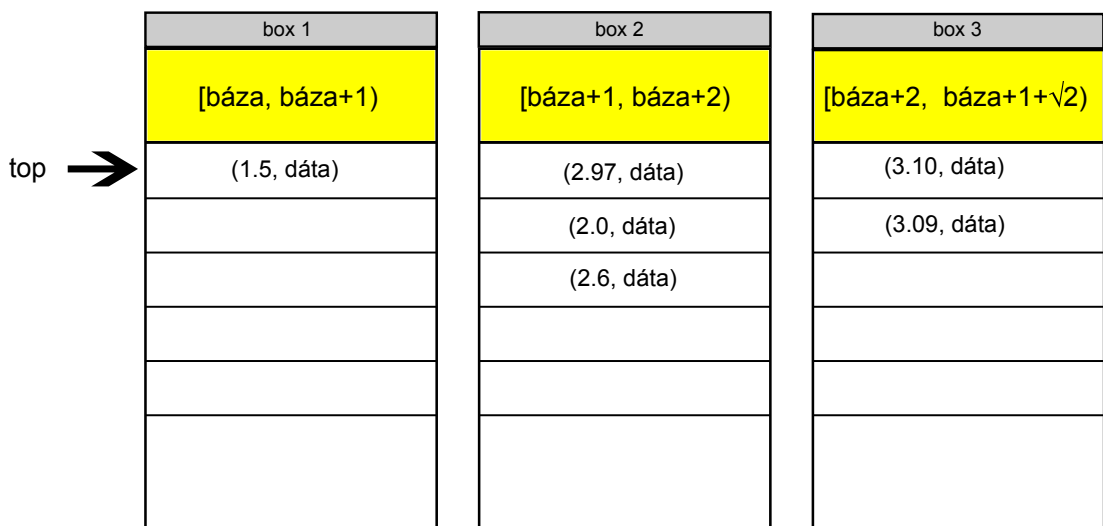
Majme haldu s tromi priehradkami (nazvime ju *BucketHeap*), pričom rozsah jednej priehradky je ostro menší, ako 1. Prvá priehradka uchováva prvky s rozsahom vzdialeností $[b, b + 1)$, druhá $[b + 1, b + 2)$ a tretia $[b + 2, b + 1 + \sqrt{2})$ pre danú bázu b . Pre jednoduchšiu implementáciu $b \in \mathbb{N}$. 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 vyhození 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 vec: 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*, ktorá definuje bázu od ktorej sa rozsahy priehradok odvíjajú. Okrem nej, uchováva tri smerníky na tri po sebe idúce priehradky priehradky a smerník na vrchol haldy, zvaný *top*.

báza = 1

1. priehradka = box 1
2. priehradka = box 2
3. priehradka = box 3



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 vrch priehradky.

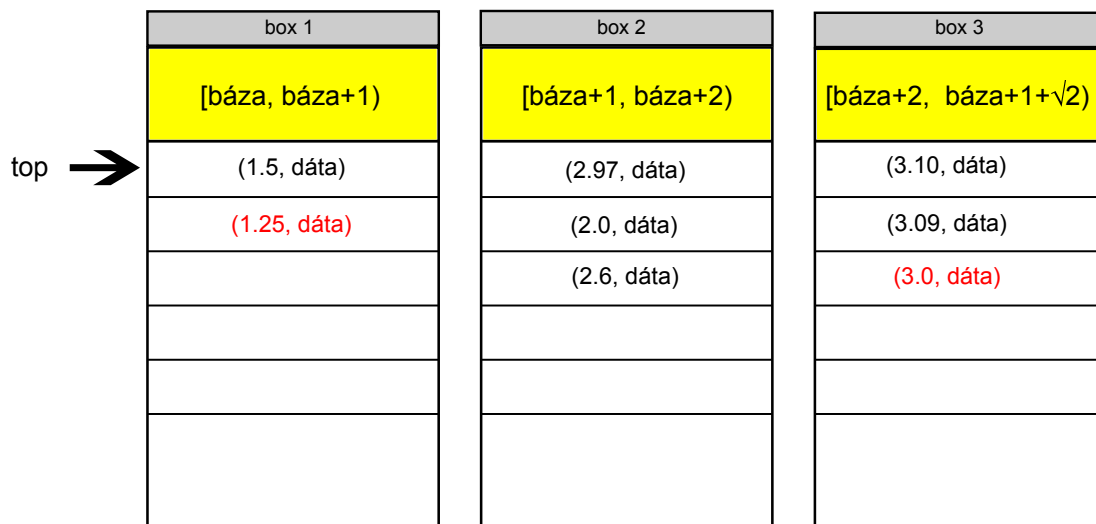
Pokiaľ sa v priehradke nachádza jediný prvok a ten chceme vybrať, tak nám inkrementácia premennej *top* v priehradke nepostačí. Musíme prehodiť priehradky. Druhú priehradku presunúť na miesto prvej, tretiu na miesto druhej a prvú umiestniť nakoniec. Vid' obrázok 2.4. Nakoniec musíme zvýšiť bázičku vzdialenosť.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

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

báza = 1

1. priehradka = box 1
2. priehradka = box 2
3. priehradka = box 3

element1 = (3.0, dáta)
 element2 = (1.25, dáta)
 insert (element1)
 insert (element2)



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

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. \square

2.3 A*

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

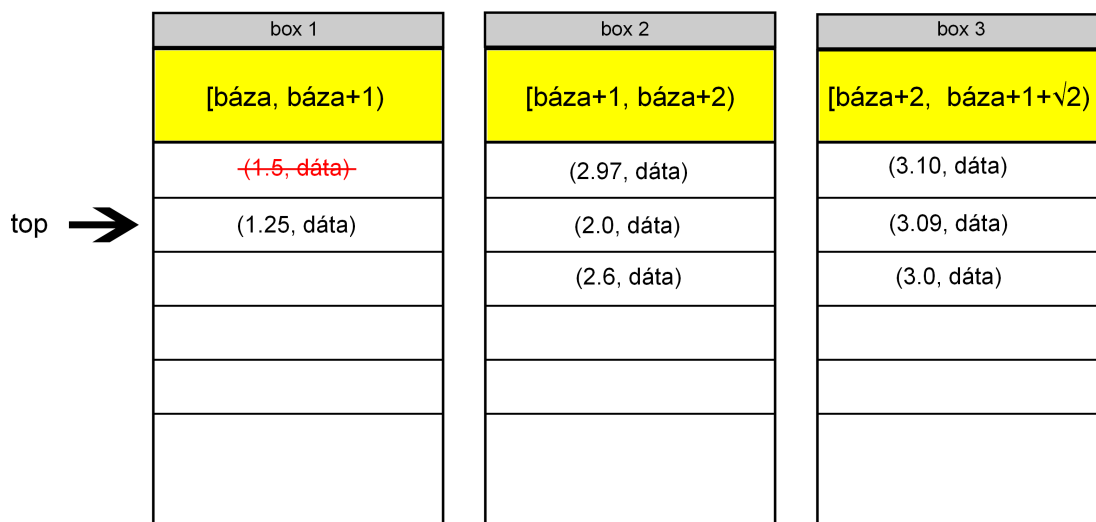
Tento algoritmus vychádza z Dijkstrovho algoritmu a je mu veľmi podobný. Hlavný rozdiel medzi týmito algoritmami je, že kým Dijkstrov algoritmus vyberá z haldy vrcholy s neklesajúcou vzdialenosťou $d(v)$ od počiatku, tak algoritmus A* vyberá prvky s neklesajúcou vzdialenosťou $f(v) := d(s, v) + h(v, t)$, 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.

báza = 1

pop()

1. priehradka = box 1
2. priehradka = box 2
3. priehradka = box 3



Obr. 2.3: Vybranie prvku — prvok na ktorý ukazuje smerník v prvej priehradke a inkrementujeme ho.

2.3.1 Heuristická funkcia

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

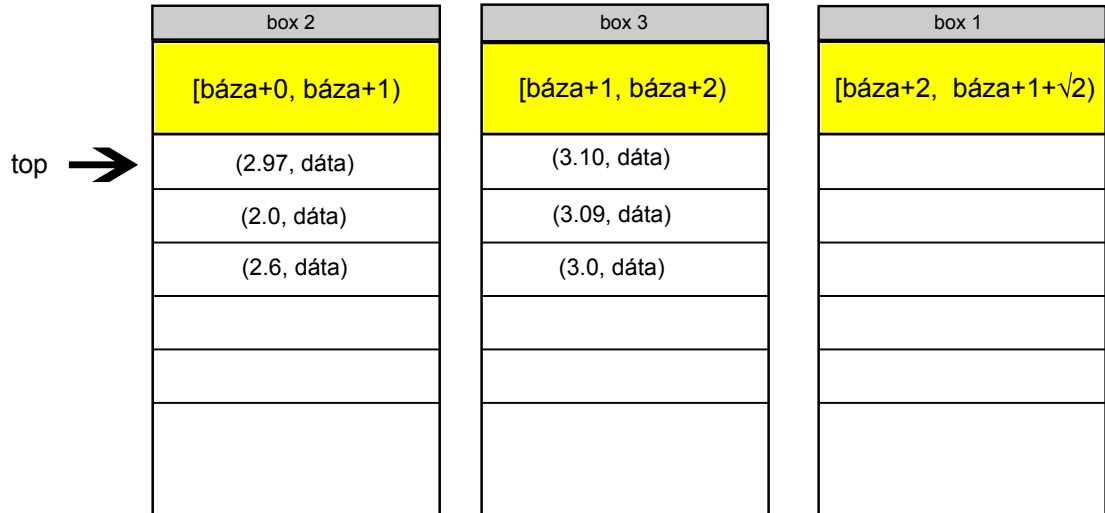
Najčastejšie heuristické funkcie sú tieto:

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

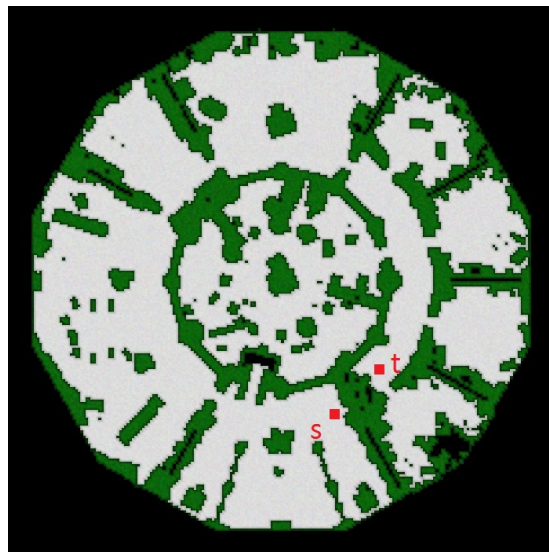
Euklidovská vzdialenosť Euklidovská vzdialenosť je najjednoduchšie implementovateľná heuristická funkcia. Na väčšine jednoduchých grafov s malým počtom prekážok vracia dobré výsledky. Problém nastáva na grafoch, kde začiatok a koniec cesty sú geometricky blízko seba, hoci ich skutočná vzdialenosť je veľká. Príklad vidíme na obrázku 2.5. TODO?? pozor na blbe rozdelenie obdĺžnikov - ja to ratam aj cez sikme hrany atd atd

báza = 2 (báza + 1) pop()

1. priehradka = box 2
2. priehradka = box 3
3. priehradka = box 1



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



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

Landmarks a trojuholníková nerovnosť Nevýhoda použitia euklidovskej heuristickej funkcie je na obecných mapách 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 $O(kn)$, kde n značí počet vrcholov grafu.

TODO?? ako to funguje...

Možnosti voľby landmarkov Pri voľbe landmarkov sú 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.*

TODO??potencialy euklid landmarks landmark selection problem s priehradkami

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 zavedme definíciu *mriežkového grafu bez prekážok*.

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

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

Majme mriežkový graf bez prekážok a hľadáme 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 Nájsť 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 počiatočok}
2: while  $x_s \neq x_t \vee y_s \neq y_t$  do
3:   if  $x_s < x_t$  then
4:      $x_s \leftarrow x_s + 1$ 
5:   else if  $x_s > x_t$  then
6:      $x_s \leftarrow x_s - 1$ 
7:   end if
8:   if  $y_s < y_t$  then
9:      $y_s \leftarrow y_s + 1$ 
10:  else if  $y_s > y_t$  then
11:     $y_s \leftarrow y_s - 1$ 
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 priamočiario, 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 $O(\max(dx, dy))$. 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

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 obe 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 4. *Ekvivalenčnú triedu predstavuje obdĺžnik a množinu predstavuje množina vrcholov grafu. Alternatívne sa môžeme na úlohu pozeráť ako na problém farbenia n guľčiek pomocou k farieb.*

Zapišme túto úlohu formálne. Majme n -prvkovú množinu $Prv = \{x_1, \dots, x_n\}$, k -prvkovú množinu ekvivalenčných tried $Ek = \{ek_1, \dots, ek_k\}$, veľkosť triedy $||ek_i||$ označme k_i a zaveďme funkciu $f: Prv \rightarrow Ek$ 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, môžeme 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, môžeme ho vynechať.

Maximalizujeme teda 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 roznásobením sme dostali nasledujúcu optimalizačnú úlohu: maximalizovať $\sum_{i=1}^k k_i^2 - k_i$ za podmienok $\sum_{k=1}^k k_i = n$, kde $k_i \in \mathbb{N}_0$.

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. Ekvivalenčné triedy musia teda byť čo najväčšie a problém sa transformuje na problém hľadania obdĺžnikov s najväčším možným obsahom. V programe tento problém rieši trieda Colorizator.

3.2.2 Nájdenie najväčšej jednotkovej podmatice

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

Slovný popis algoritmu 3: V prvom prechode maticou si u každého vrcholu zapamätáme počet jedničiek naľavo od neho, vrátane daného vrcholu. Tento prechod trvá lineárny čas.

V druhom prechode treba prejsť zaradom všetky stĺpce zľava doprava

TODO: pridaj obrazok mapy a jej dekompozície opdla farieb

Algoritmus 3 Nájdenie najväčšej jednotkovej podmatice v matici $m \times n$

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

Výstup: pravý dolný roh podmatice aj s jej rozmermi

```
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 dvojíc (riadok, nalavoOdPrvku)
10:  for riadok  $r, r \in M$  do
11:     $p \leftarrow (s, r)$ 
12:    while Zásobník je neprázdny do
13:      vyberiem prvok  $top$  z vrcholu zásobníka
14:      if  $top . \text{nalavoOdPrvku} > \text{nalavoOdPrvku}(p)$  then
15:        prvokZoZasobnika  $\leftarrow (top.r, s)$ 
16:        DĺžkaSekvencie(prvokZoZasobnika)  $= r - \text{prvokZoZasobnika}.r$ 
17:      else
18:        Vložím prvok  $\text{prvokZoZasobnika}$  do zásobníka
19:        break
20:      end if
21:    end while
22:    Do zásobníka vložím dvojicu  $(r, \text{nalavoOdPrvku}(p))$ 
23:  end for
24:  while Zásobník je neprázdny do
25:    vyberiem prvok  $top$  z vrcholu zásobníka
26:    prvokZoZasobnika  $\leftarrow (top.r, s)$ 
27:    DĺžkaSekvencie(prvokZoZasobnika)  $= m - \text{prvokZoZasobnika}.r$ 
28:  end while
29: end for
```

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 mi 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. Konkrétny popis typov máp sa nachádza v [10].

4.1.2 Testovacie kritériá

Testovacie kritériá budú podobné ako testovacie kritériá v súťaži. Budú sa však mierne líšiť, pretože algoritmus nebol naplánovaný tak, aby TODO?? (prvých 20 krokov a nezaujima, na druhú stranu ma zaujíma počet prehľadaných vrcholov...)

- Počet prehľadaných vrcholov.
- Rýchlosť nájdenia cesty.

4.1.3 Testované algoritmy

Testovať budeme nasledujúce algoritmy:

- Dijkstrov algoritmus nad binárnou haldou.
- Dijkstrov algoritmus nad priehradkovou haldou.
- A* používajúc rôzny počet landmarkov.
- A* používajú optimálny počet landmarkov a obdĺžnikovú dekompozíciu.
- TODO?? niečo ďalšie blablabla - cudzie algoritmy.

4.1.4 Kompilácia

Kód bude kompilovaný kompilátorom g++. Bude porovnaná rýchlosť behu programu pri kompilácii s týmito direktívami.

- -O1
- -O2
- -O3
- -O3 -march=native

ASK?? toto bude len pri jednom algoritme pri jednom grafe, nie? není to zbytočne???

4.1.5 Typy map a ciest

TODO?? - kratke dlhe, klukate a rovne obrazok asopn 3 map

(ASK?? poet prehľadanych vrcholov nie je kritérium, ktoré dokážem testovať na cudzích algoritmoch - musel by som do nich zahat a pridávať tam funkciu, ktorá to počíta - čo mám robiť? mám počet vrcholov testovať len na mojich algoritmoch?)

Na porovnávanie využijeme benchmark

TODO?? bibliografia styl - priezvisko, meno - alebo naopak???

5. T-maps - Vizuálna predstava

Pre názornejšiu predstavu behu algoritmu sme navrhli softvér **T-maps**, ktorý beh algoritmov ilustruje graficky.

5.1 Použitie

Program T-maps Na začiatku do neho nahrajeme mapu, nad ktorou algoritmus beží a taktiež dáta tohto algoritmu (prehľadané vrcholy a najkratšiu cestu) a program tieto hodnoty graficky znázorní. Medzi možnosti programu patrí export mapy a viditeľného poľa do súboru. TODO?? uzivatelska dokumentacia Možno popis zaujimavej casti

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 algoritmu*. ITI Series, Prague 2007. 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*, 2013. 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.

Zoznam tabuliek

Zoznam použitých skratiek

Prílohy