Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vyhľadávanie spojení mestskou hromadnou dopravou

Bakalárska práca

2018
Alojz Stúpal

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vyhľadávanie spojení mestskou hromadnou dopravou

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Michal Forišek, PhD.

Bratislava, 2018
Alojz Stúpal

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Alojz Stúpal
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Vyhľadávanie spojení mestskou hromadnou dopravou
Finding connections using public transport

Anotácia: Práca sa zaoberá vyhľadávaním a zobrazovaním možností cesty MHD.

Cieľ: Práca má nasledujúce ciele:
1. Spracovať prehľad základných algoritmov vhodných na vyhľadávanie spojenia v grafikone MHD.
2. Implementovať softvér umožňujúci takéto vyhľadávanie a otestovať jeho funkčnosť na reálnych dátach.
3. Pokúsiť sa navrhnúť a implementovať možné vylepšenia, ako napr.: vyhľadávanie cesty bod-bod namiesto zastávka-zastávka; vizualizácia možných alternatívnych trás; výpočet dodatočných informácií, napr. očakávaného meškania v prípade nestihnutia prestupu.

Vedúci: RNDr. Michal Forišek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 07.11.2017

Dátum schválenia: 08.11.2017 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem školiteľovi, RNDr. Michalovi Foriškovi, PhD., za vynaložené úsilie, ktoým mi ustavične pomáhal. Ďakujem i všetkým ostatným, ktorí mi akokoľvek pomohli pri písaní tejto práce.

Abstrakt

V práci najprv popisujeme algoritmy použiteľné pri vyhľadávaní spojení v grafikone mestskej hromadnej dopravy. Následne argumentujeme, ktorý je na tento účel najlepší. Implementujeme program, ktorý realizuje takéto vyhľadávanie zapomoci zvoleného algoritmu. Implementujeme vylepšenie algoritmu - zobrazovanie alternatívnych ciest.

Kľúčové slová: grafy, najlacnejšie cesty v grafe, vyhľadávanie spojení MHD, Dijkstrov algoritmus

Abstract

We will describe algorithms that are usable when searching for links in public transport. After that, we will give arguments about which one is the best for this purpose. With chosen algorithm we will implement a program that will perform such a search. We will implement an improvement in algorithm - displaying alternative paths.

Keywords: graph, shortest paths in graph, search for public transport links, Dijkstra's algorithm

Obsah

Zoznam obrázkov

Zoznam tabuliek

Úvod

Cieľom tejto práce je spracovať informácie a vytvoriť prehľad algoritmov, ktoré sú použiteľné pri vyhľadávaní spojenia v grafikone hromadnej dopravy a následne stručne popísať ich výhody i nedostatky pri ich využití na takéto vyhľadávanie. Praktickou stránkou práce je implementácia algoritmu, ktorý sme zhodnotili, že je najvhodnejší na vyhľadávanie a zakomponovanie algoritmu do aplikácie vhodnej na používanie. Ďalším krokom je rozšírenie programu o dodatočnú funkcionalitu, konkrétne vizualizáciu alternatívnych trás.

V kapitole 1 oboznámime čitateľa so základnými pojmami, ako je napríklad graf či cesta, ako i s ďalšími potrebnými definíciami, ktoré budú využívané v ostatných kapitolách.

V kapitole 2 popíšeme algoritmy aplikovateľné na grafové štruktúry a využiteľné pri vyhľadávaní spojení hromadnej dopravy. Pre každý algoritmus uvážime, prečo je vhodný pre takéto vyhľadávanie a prečo nie.

V kapitole 3 sa pozrieme na softvér, ktorý sme implementovali. Uvedieme jeho možnosti a funkcie, popíšeme, ako ho ovládať a ukážeme i nejaké príklady z jeho použitia.

V kapitole 4 odhalíme, aké úvahy a myšlienky nás sprevádzali, aké postupy, algoritmy či štruktúry sme zvolili pri implementácií opísaného softvéru.

Kapitola 1

Grafy

V tejto kapitole uvedieme zopár definícií týkajúcich sa grafov, opíšeme niektoré grafové štruktúry, prípadne zavedieme rôzne názvy, ktoré budeme v ďalších častiach práce potrebovať.

1.1 Definícia grafu

Začneme so základnou definícou - s definovaním pojmu *graf*. Tú prevezmeme od Reinharda Diestela [6, kapitola 0.1]:

Graf je dvojica $G = (V, E)$ disjunktných množín, kde prvky E sú dvojprvkové podmnožiny V. Prvky V sú *vrcholy* (prípadne *uzly* alebo *body*) grafu G, prvky E sú jeho hrany.

Vytvorený objekt by sa mal korektne nazývať *neorientovaný graf*. Avšak my, ako aj mnohí iní, od tohto pomenovania upustíme a budeme pre jednoduchosť používať len pojem *graf*.

Hranu $\{x, y\}$, kde $x, y \in V$, budeme zasa zvyčajne označovať aj ako (x, y) alebo xy.

1.2 Základné pojmy týkajúce sa grafov

V nasledujúcej podkapitole sú uvedené definície rôznych pojmov, ktoré charakterizujú a popisujú vlastnosti grafových štruktúr.

Rád grafu definujeme ako počet vrcholov grafu G.

Budeme hovoriť, že vrchol v je *incidentný* s hranou e, ak $v \in e$. Čiže, ak hrana e

$= xy$, oba vrcholy x aj y nazveme incidentnými s hranou e. Ak sú dva rôzne vrcholy incidentné s tou istou hranou, budeme ich volať jej *koncovými vrcholmi*.

Dva vrcholy $x, y \in V$ sú *susedné*, ak xy je hrana v G. Dve hrany $e, f \in E; e \neq f$ sú *susedné*, ak majú spoločný vrchol.

Stupňom vrchola $v \in V$ nazveme počet hrán incidentných s v. Alebo inak povedané, je to počet vrcholov, ktoré sú susedné s vrcholom v.

1.2.1 Typy grafov

Teraz uvedieme niektoré základné typy grafov.

Prázdny graf je graf $G = (\emptyset, \emptyset)$. Označenie zjednodušíme na $G = \emptyset$.

Pojmom *Triviálny graf* budeme označovať graf s rádom 0 alebo 1.

Kompletným grafom nazveme taký graf, ktorého všetky vrcholy sú navzájom susedné. Uvažujme n vrcholov, potom kompletný graf na týchto vrcholoch označíme K_n. Na predstavu môže poslúžiť príklad kompletného grafu troch vrcholov K_3, čo je trojuholník.

Majme graf G. Ak každý jeho vrchol má rovnaký stupeň k, potom tento graf nazveme *k-regulárnym*. V definícií môžeme upustiť od počtu vrcholov a nazvať vzniknutý objekt iba *regulárnym*.

1.2.2 Vzťahy medzi grafmi

Nech $G = (V, E)$ a $G' = (V', E')$ sú grafy. Označíme $G \cap G' = (V \cap V', E \cap E')$ a $G \cup G' = (V \cup V', E \cup E')$. Ak prienik týchto dvoch grafov je prázdny graf $G \cap G' = \emptyset$, potom G a G' sú *disjunktné*. Ak $V' \subseteq V$ a $E' \subseteq E$, potom hovoríme, že G' je podgraf G (alebo G je nadgraf G', či G obsahuje G') a píšeme $G' \subseteq G$.

1.3 Cesty a cykly

Nech $G = (V, E)$ je graf. Definujme *cestu* ako neprázdnu striedavú postupnosť $v_0e_0v_1e_1...e_{k-1}v_k$, kde $v_i \in V, e_j \in E$ pre $i = 1, 2, ..., k$ a $j = 1, 2, ..., k-1$, pričom $e_i = v_iv_{i+1}$ pre všetky

$i < k$ a všetky v_i sú navzájom rôzne. Neformálne povedané, cesta spája (pomocou hrán) dva vrcholy grafu G, pričom sa v nej použité vrcholy nesmú opakovať. Počet hrán cesty sa nazýva aj *dĺžka* cesty.

Majme graf $G = (V, E)$. Nech $P = v_0e_0v_1e_1...e_{k-1}v_k$ je cesta v G, pričom $k \geq 2$ a v grafe G existuje hrana v_0v_k. Potom hovoríme, že graf G obsahuje *cyklus* alebo tiež *kružnicu* C. Cyklus definujeme opäť ako neprázdnu striedavú postupnosť vrcholov a hrán, v tomto prípade $C = v_0e_0v_1e_1...e_{k-1}v_ke_k$, kde $e_k = \{v_0, v_k\}$.

Sled v grafe G je názov pre neprázdnu striedavú postupnosť $v_0e_0v_1e_1...e_{k-1}v_k$ vrcholov a hrán v G, pričom $e_i = v_iv_{i+1}$ pre všetky $i < k$. Sled teda, na rozdiel od cesty, môže obsahovať rovnaké vrcholy viac krát. Preto sa dá povedať, že sled je cesta, v ktorej sa môžu vrcholy opakovať. Ale aj naopak: sled, v ktorom sú vrcholy navzájom rôzne, je cestou.

Sled, v ktorom sú hrany navzájom rôzne, sa nazýva *ťah*.

1.4 Súvislosť

Ak pre ľubovoľné dva vrcholy grafu G existuje cesta, tento graf nazveme *súvislým*.

Komponent grafu G je súvislý podgraf grafu G taký, že nie je obsiahnutý v žiadnom väčšom súvislom podgrafe grafu G. Napríklad, súvislý graf má práve jeden komponent, a to samého seba.

Hrana $e \in E$ grafu G sa volá *most*, ak graf G má menší počet komponentov v porovnaní s grafom G bez hrany e. Čiže, po odstránení hrany z grafu pribudne práve jeden komponent.

Vrchol $v \in V$ grafu G sa nazýva *artikulácia*, ak počet komponentov grafu G je menší ako ich počet po odstránení v z grafu G. Teda, ak po odstránení vrchola z grafu pribudne aspoň jeden komponent.

1.5 Kostry, stromy a lesy

Nech $G = (V, E)$ je súvislý graf. *Kostrou* grafu nazveme taký graf $G' = (V', E'), G' \subseteq G$, pre ktorý platí $V' = V$ a navyše medzi každými dvoma vrcholmi existuje práve jedna cesta. Ak by sme grafu G postupne odnímali hrany tak, aby sme nenarušili jeho súvislosť, tak po odstránení všetkých takýchto hrán by sme získali kostru grafu G. Z tejto konštrukčnej definície vyplýva, že všetky hrany v kostre grafu sú mostami. Je z nej taktiež zrejmé, že každá kostra grafu je súvislá.

Kostrový les je taký graf G, pre ktorý platí, že každý z jeho komponentov je kostrou.

Acyklický graf (to jest taký, ktorý neobsahuje cyklus), ktorý je navyše súvislý, nazveme *stromom*. Všetky vrcholy stromu so stupňom 1 sú jeho *listy*. Je vhodné si povšimnúť, že, ako pri kostre, všetky hrany sú mostami. Taktiež je zaujímavé uvažovať o rozdiele medzi stromom a kostrou grafu. Možno nahliadnuť, že jediným rozdielom medzi nimi je, že strom je grafom sám o sebe, zatiaľ čo o kostre hovoríme len v súvislosti v grafom, z ktorého vznikla.

Les je acyklický graf. Takže každý jeho komponent je stromom.

1.6 Špeciálne grafové štruktúry

Graf, ktorého prvky sú ohodnotené číslom určujúcim cenu, prípadne výhodnosť prechodu cezeň, nazveme *ohodnotený graf*. *Hranovo ohodnotený graf* je graf s funkciou $w : E \to R$. Teda pre ľubovoľnú hranu e existuje číslo $w(e)$, ktoré nazveme *hodnotou hrany*, prípadne *cenou hrany*. Z praktického hľadiska je výhodné zadefinovať aj *kladne hranovo ohodnotený graf*. A to je taký, že $w(e) > 0$ pre všetky hrany. *Vrcholovo ohodnotený graf* je graf s funkciou $w : V \to R$. Teda pre ľubovoľný vrchol v existuje číslo $w(v)$.

Orientovaný graf je dvojica $G = (V, E)$, kde každý prvok $xy \in E$ vyjadruje usporiadanú dvojicu. Neformálne, ak je daná hrana xy, ale nie yx, existuje cesta dĺžky 1 z vrcholu x do y, ale z vrcholu y do x takáto cesta nejestvuje.

Rozdiel medzi orientovanými a neorientovanými grafmi je, ako názov napovedá v existencií orientácií hrán. Ak by sme túto skutočnosť chceli vyjadriť formálne, povedali by sme, že zatiaľ čo v neorientovanom grafe obsahuje množina E dvojprvkové množiny, v orientovanom sa skladá z usporiadaných dvojíc.

1.6.1 Cesty a cykly pre orientované grafy

Definícia cesty i cyklu pre orientované grafy je rovnaká ako pre neorientované. Rozdiel je len v jej použití na orientovaných grafoch. Pre úplnosť ich ale v stručnej podobe uvádzame.

Nech $G = (V, E)$ je orientovaný graf. Nech $P = v_0e_0v_1e_1...e_{k-1}v_k$, kde $v_i \in V, e_j \in E$ pre $i = 1, 2, ..., k$ a $j = 1, 2, ..., k-1$, pričom $e_i = v_iv_{i+1}$ pre všetky $i < k$ a všetky x_i sú po dvoch rôzne. Potom P nazývame *cestou* v grafe G. Ak $k \geq 3$ a vrcholy sú rôzne až na rovnosť $x_0 = x_k$, potom nazveme P *cyklom*, prípadne *kružnicou*. [1, kapitola 1.4]

Kapitola 2

Algoritmy na grafoch

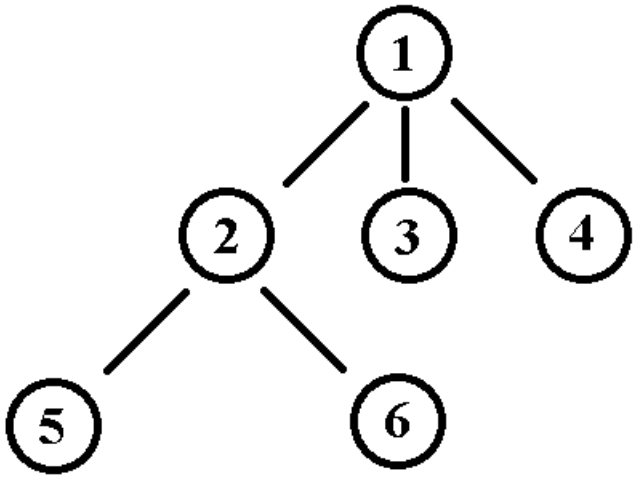
Táto kapitola obsahuje popisy algoritmov, ktoré sú aplikovateľné na grafové štruktúry a navyše využiteľné pri vyhľadávaní spojení v hromadnej doprave. Formálnu charakterizáciu algoritmu zväčša sprevádza i jeho voľnejšia interpretácia, ktorá má za účel uľahčiť porozumenie čitateľom, ktorí sa s daným algoritmom doposiaľ nestretli. Naviac, pri algoritmoch sú uvedené výhody, respektíve nedostatky, pri ich použití na nami vytýčený cieľ. V hojnom počte budeme využívať definície z kapitoly 1.

2.1 Prechádzanie vrcholmi grafu

Stručne popíšeme základné algoritmy na prechádzanie vrcholov grafu. Tie však nie sú priveľmi využiteľné pri vyhľadávaní spojení MHD. Môžu nám ale významne pomôcť k prechádzaniu objemných súborov dát, ktoré bude sieť hromadnej dopravy isto obsahovať.

2.1.1 Prehľadávanie do šírky

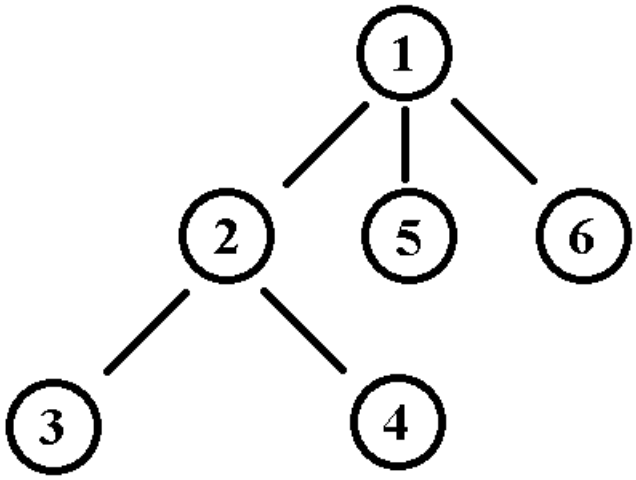
Prehľadávanie do šírky alebo skrátene BFS (z anglického Breadth-first search), dokáže pejsť cez všetky vrcholy grafu za pomoci fronty (alebo radu). Algoritmus si do nej ukladá susedov spracúvaných vrcholov, a tak prehľadá celý graf, pričom v každej iterácií prejde o jednu hranu viac vzdialené vrcholy od počiatočného.



Obr. 1: Príklad práce BFS na strome

2.1.2 Prehľadávanie do hĺbky

Prehľadávanie do hĺbky, skrátene DFS (z anglického Depth-first search), prechádza taktiež cez všetky vrcholy grafu. Obvykle sa v algoritme na všetkých susedov vrchola zavolá postupne rekuzívne ten istý algoritmus prehľadávania.



Obr. 2: Príklad práce DFS na strome

Je vhodné poznamenať, že na obrázku 2 je ukázané len jedno z možných poradí práce s vrcholmi. Samozrejme, vrcholy budú vždy navštívené vo vyobrazenom poradí, avšak práca s nimi môže byť vykonaná až tromi spôsobmi. Prvý z nich súhlasí s usporiadaním na obrázku a nazýva sa *preorder*. Ďalšie dva sú *inorder* a *postorder*.

2.1.3 Zhrnutie

Obi dva algoritmy majú očividnú lineárnu časovú zložitosť ($O(n)$), nakoľko sa pozrú na každý vrchol grafu práve raz.

Ako sme už spomínali, v týchto algoritmoch nevidíme potenciál pri ich využití na uskutočnenie nášho cieľa. Ich znalosť však môže byť veľmi prínosná pri prechádzaní štruktúr našich grafov.

2.2 Najlacnejšie cesty v grafe

Pri vyhľadávaní spojení mestskou hromadnou dopravou sa zdá byť veľmi racionálne zaoberať sa nachádzaním najlacnejších ciest v grafikone MHD. Dovoľujeme si tak tvrdiť, pretože najväčší dôraz cestujúcich je kladený práve na čo najskorší príchod do požadovanej lokality. Pre úplnosť ešte dodávame, že cena cesty, ako je asi zrejmé, je súčet hodnôt hrán (respektíve vrcholov), ktoré daná cesta obsahuje. Z doposiaľ uvedeného vyplýva, že pri našich úvahách budeme používať hranovo ohodnotené grafy, kde pridelenou hodnotou bude čas medzi zastávkami. Navyše musíme nejako ošetriť i vrcholy grafu - zastávky, keďže na nich zvyčajne čakáme na prestup na ďalší spoj. Od tejto myšlienky však spočiatku upustíme a budeme sa jej venovať až neskôr. A posledná úvaha - hranami v našom grafe sú linky MHD, sme preto nútení použiť orientované grafy.

Ak uvažujeme vyhľadávanie najlacnejších ciest v grafe, musíme si najprv uvedomiť, čo presne je našim cieľom. Výsledok, ktorý chceme dosiahnuť ako výstup algoritmu, má byť najlacnejšia cesta z počiatočného bodu do koncového. Avšak znalosť problematiky algoritmov na grafových štruktúrach nám ponúka riešenia iného, komplexnejšieho problému s asymptoticky rovnakou časovou zložitosťou. Týmto problémom je vyhľadávanie najlacnejších ciest z počiatočného vrcholu do všetkých ostatných vrcholov. Ľahko nahliadnuť, že riešením tohto problému dostaneme odpoveď aj na našu počiatočnú otázku. Preto sa v nasledujúcej časti budeme zaoberať algoritmami riešiacimi túto úlohu.

Ľahko spozorovať, že by mohlo byť v určitých situáciách výhodné vypočítať ceny a nájsť najlacnejšie cesty pre všetky dvojice vrcholov. Hlavne, keď graf obsahuje malé množstvo vrcholov - zastávok. Mnoho miest má len malú sieť mestskej hromadnej dopravy, a teda by bolo rozumné vypočítať všetky potrebné údaje naraz na začiatku, a potom, pri prijímaní dotazu na vyhľadanie spojenia, jednoducho vypísať už vypočítané

hodnoty. Z tohto dôvodu uvedieme i algoritmy riešiace tento problém.

2.2.1 Dijkstrov algoritmus

Holandský informatik, po ktorom je tento algoritmus pomenovaný, dokázal, okrem iného, vyriešiť aj nami nastolený problém. V jeho riešení je ale potrebné, aby boli ceny hrán grafu nezáporné reálne čísla, čo súhlasí s našou predstavou aplikovania algoritmu na grafikon MHD. Algoritmus je navrhnutý tak, že dostane ako vstup graf $G = (V, E)$, počiatočný vrchol v_0 a hodnotiacu funkciu $h : V \times V \to R_0^+$. Predpokladáme, že ak hrana $uv \notin E$, potom $h(u, v) = \infty$, ďalej, že $h(u, u) = 0$ a nakoniec, že funkciu h je možné vypočítať v konštantnom čase $O(1)$. Máme taktiež jeden predpoklad na vrcholy grafu, a to, že sú reprezentované celými číslami $1, 2, ..., k$ (kde k je počet vrcholov grafu). Nakoniec, výsledkom algoritmu bude pole čísel D, v ktorom bude pre každý vrchol $v \in V$ vypočítaná hodnota $D[v]$, čo je cena najlacnejšej cesty z počiatočného vrchola v_0 do vrchola v.

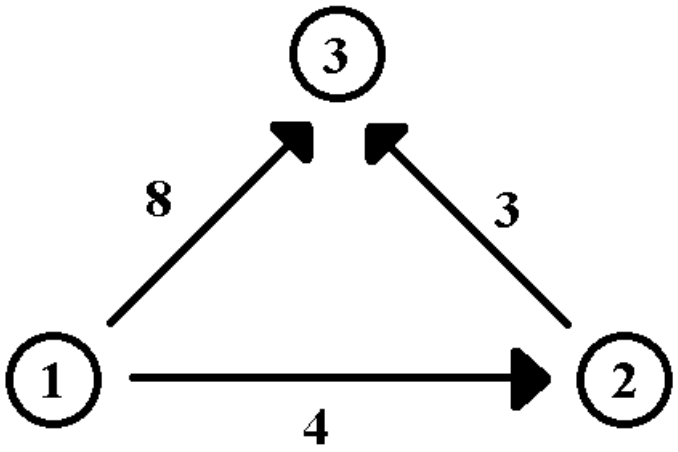
Tieto požiadavky, predpoklady, ba i vstup a výstup funkcie sú uvažované v horeuvedenom tvare len preto, aby sme mohli predviesť implementáciu algoritmu od Pavla Ďuriša, ktorú možno nájsť aj v jeho knihe [19, kapitola 2.2.1].

begin
$S \leftarrow \{v_0\}$
$D[v_0] \leftarrow 0$
pre každý vrchol $v \in V \setminus \{v_0\}$: $D[v] \leftarrow h(v_0, v)$
while $S \neq V$ **do begin**
vyber $w \in V \setminus S$ *taký, že hodnota* $D[w]$ *je minimálna*
$S \leftarrow S \cup \{w\}$
pre každý $v \in V \setminus S$:
$D[v] \leftarrow min\{D[v], D[w] + h(w, v)\}$
end
end

Algoritmus 1: Dijkstrov algoritmus

Časová zložitosť Dijkstrovho algoritmu je $O(|V|^2)$, čo je dokázané spolu s jeho ko-

rektnosťou v už uvedenom zdroji od Pavla Ďuriša [19, kapitola 2.2.1].



Obr. 3: Orientovaný graf s ohodnotenými hranami

Priebeh Dijkstrovho algoritmu vysvetlíme na jednoduchom príklade. Máme orientovaný graf ako je zobrazený na obrázku 3. Našou úlohou je nájsť najkratšiu cestu z vrcholu 1 do ostatných vrcholov. Učiníme tak teda za pomoci Dijkstrovho algoritmu. Čiže, najprv si vytvoríme pole, označme ho D, v ktorom si pre každý vrchol budeme pamätať hodnotu doň najkratšej cesty z vrcholu 1. Na počiatku bude táto hodnota pre všetky vrcholy nekonečno, respektíve nejaká jeho rozumná náhrada. My budeme používať nekonečno. Nastavíme $D[1] = 0$. Začíname teda s počiatočným vrcholom 1. Zoberieme jeho všetkých zatiaľ nenavštívených susedov (vrcholy 2 a 3), a porovnáme hodnoty takto: najprv pre vrchol 2, zistíme či je jeho hodnota v poli D (to jest nekonečno) menšia ako hodnota vrchola, z ktorého vychádzame, plus hodnota hrany (to jest $0 + 4$). To ale nie je pravda, takže prepíšeme hodnotu $D[2]$ na 4. Pre vrchol 3 zasa porovnávame nekonečno s hodnotou $0 + 8$ alebo aj, ako je v algoritme 1 naznačené, minimum z týchto dvoch hodnôt zapíšeme do $D[3]$. Vrchol 1 nemá viac susedov, takže ho označíme ako navštívený a pokročíme. Ďalším vrcholom v poradí bude ten s najmenšou hodnotou $D[i]$ spomedzi doposiaľ nenavštívených, čiže vrchol 2. Ten má iba jedného ešte nenavštíveného suseda, a to vrchol 3. Do $D[2]$ preto zapíšeme minimum z hodnôt 8 a $4 + 3$. Vrchol 2 je týmto vybavený a označený za navštívený. Zostáva posledný vrchol - vrchol 3. Väčšina implementácií Dijkstrovho algoritmu pri dosiahnutí finálneho vrcholu končí v snahe urýchliť vyhľadávanie medzi dvoma vrcholmi. Ak však túto podmienku nezakomponujeme do nášho algoritmu, program zráta hodnoty najkratších ciest z daného vrcholu do všetkých ostatných vrcholov. V našom prípade nie je veľmi podstatné, ktorú variantu zvolíme, keďže algoritmus tak či tak vo vrchole 3 skončí, nakoľko ten už nemá žiadne incidentné hrany a v grafe už nejestvujú ďalšie ne-

navštívené vrcholy. Dosiahli sme teda výsledok: najkratšie cesty z vrcholu 1 sú uložené v poli D. Najkratšia cesta do vrcholu 1 je $D[1] = 0$, do vrcholu 2 je $D[2] = 4$ a pre vrchol 3 je odpoveď $D[3] = 7$.

Týmto príkladom sme chceli jemne ozrejmiť postup pracovania algoritmu, keďže pre tých, čo sa s ním ešte nestretli, ho môže byť dosť namáhavé z pseudokódu 1 vybadať.

Keďže je našim cieľom nie ani tak zistiť cenu najlacnejšej cesty, čo je hlavnou úlohou popísaného algoritmu, ale skôr takéto cesty nájsť, Dijkstrov algoritmus by sme potrebovali jemne modifikovať, a to tak, aby sme vedeli spätne zrekonštruovať nájdené cesty. Teda so znalosťou konečného vrchola by sme chceli vedieť vygenerovať postupnosť vrcholov, cez ktoré sme sa do finálneho vrchola dostali. Budeme si preto pri každom vrchole pamätať informáciu, z ktorého vrchola sme sa doň dostali.

Dijkstrov algoritmus sa zdá byť najvhodnejším kandidátom pre účely našej práce. Jeho časová zložitosť je príjemná, implementácia nenáročná a je dosť ľahké sa v nej orientovať, takže si ju budeme môcť poľahky modifikovať, prispôsobiť našim potrebám. Budeme teda schopný vypísať dodatočné informácie pre používateľov aplikácie, prípadne vyhľadávanie spresniť či rozšíriť podľa ich potrieb.

Vylepšenie pamätaním si susedov

Je dosť ľahké všimnúť si, že v momente Dijkstrovho algoritmu, keď máme vybraný vrchol, s ktorým chceme skontrolovať cenu cesty do ostatných vrcholov, nemusíme prechádzať cez všetky vrcholy, ale len cez tie, s ktorými daný vrchol susedí. Samozrejme, ak pracujeme na kompletnom grafe, zlepšenie bude nulové, navyše, počítanie susedov pre každý vrchol iba predĺži čas bežania algoritmu. Časová zložitosť Dijkstrovho algoritmu sa teda implementáciou tohto vylepšenia nezmení. Avšak ak budeme pracovať na grafoch, o ktorých vieme, že sú riedke, toto vylepšenie bude naozaj prínosné. Ostáva vyriešiť časovú zložitosť vypočítania susedov pre každý vrchol. Nakoľko sa ale tieto výsledky dajú predpočítať ešte pred spustením algoritmu, napríklad pri vytváraní grafu, časová zložitosť Dijkstrovho algoritmu bude stále rovnaká.

V našom prípade aplikácie algoritmu na nachádzanie najlacnejších ciest v grafikone sa teda toto vylepšenie stáva vynikajúcim, nakoľko graf liniek hromadnej dopravy zvykne byť veľmi riedky.

Vylepšenie pridaním haldy

Dijkstrov algoritmus, ako sme ho popísali, sa ale v tejto podobe nevyužíva, nakoľko je známa lepšia, asymptoticky rýchlejšia implementácia. Problémom pôvodného algoritmu je vyhľadávanie minima v poli vrcholov, ktoré sa deje v každej iterácií algoritmu. Toto vyhľadávanie sa uskutočňuje v asymptotickom čase $O(|V|)$. Je však známe, že použitím dátovej štruktúry *halda* sa problém dá riešiť v časovej zložitosti $O(\log |V|)$. Implementácia s použitím haldy je preto rozšírenejšia ako pôvodná verzia. Keďže ale môže byť algoritmom aj tak spracovaná každá jedna hrana grafu, musí byť aj tento aspekt zahrnutý do časovej zložitosti algoritmu. Možno je vhodné podotkúť, že maximálny počet hrán je $|V|^2$ - medzi každými dvoma vrcholmi jedna, a tak časová zložitosť pôvodného algoritmu už zahŕňa možnosť prejdenia všetkých hrán. Celková časová zložitosť Dijkstrovho algoritmu s použitím haldy je teda $O((|V| + |E|) \log |V|)$.

2.2.2 Bellman–Fordov algoritmus

Ďalším algoritmom na výpočet najlacnejších ciest v grafe je Bellman–Frodov algoritmus. Hoci ako prvý prišiel s týmto riešením Alfonso Shimbel v roku 1955, algoritmus bol pomenovaný po Fordovi, ktorý ho publikoval v roku 1956 a po Bellmanovi, ktorého publikácia je z roku 1958. Ten istý algoritmus zverejnil i Moore v roku 1957, a preto niekedy môžeme počuť aj o Bellman–Ford–Moorovom algoritme.

Samotný algoritmus, na rozdiel od Dijkstrovho, dokáže pracovať i so záporne ohodnotenými hranami. Avšak predpokladá sa, že graf neobsahuje záporne cykly, to jest cykly, ktorých celková cena má zápornú hodnotu. Ak by v grafe existoval aspoň jeden, mohli by sme po ňom prejsť ľubovoľne veľa krát, čím by sme ustavične znižovali cenu cesty do viacerých vrcholov. Bellman–Fordov algoritmus je však navrhnutý tak, že zakaždým vykoná konečný počet krokov, takže i pri grafe obsahujúcom záporný cyklus skončí, nezacyklí sa, iba jeho výsledné hodnoty nebudú pravdivé. Navyše, mnohé implementácie zahŕňajú i detekciu takéhoto cyklu a pri jeho nájdení vyhlásia chybu. Naša implementácia takúto kontrolu obsahovať nebude, nakoľko nie je potrebná pre vysvetlenie funkcionality a fungovanie algoritmu.

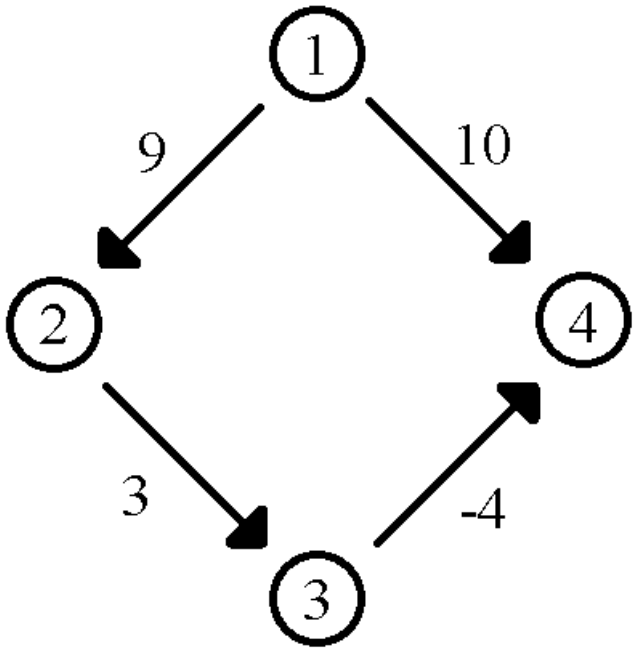
Náš algoritmus predpokladá, že na vstupe dostane graf $G = (V, E)$ neobsahujúci záporné cykly, hodnotiacu funkciu $h : V \times V \to R$ a počiatočný vrchol v_0. Ďalšie predpoklady sú, ako u Dijkstru, že ak $uv \notin E$, potom $h(u, v) = \infty$, taktiež $h(u, u) = 0$ a napokon, že funkciu h je možné vypočítať v čase $O(1)$. Aby sa nám ľahšie pracovalo, opäť povedzme, že vrcholy na vstupe musia byť reprezentované celými číslami $1, 2, ..., n$, kde n je počet vrcholov grafu. Našim výsledkom bude pole čísel D, v ktorom

bude pre každý vrchol $v \in V$ vypočítaná cena najlacnejšej cesty z počiatočného vrchola v_0 do v, označená $D[v]$. Nasledovná implementácia je zostavená s pomocou knihy od Bannistera a Eppsteina [2].

$D[v_0] \leftarrow 0$
pre každý vrchol $v \in V \setminus \{v_0\}$: $D[v] \leftarrow \infty$
for $i \leftarrow 1$ to $n-1$ **do**
  for each edge wv in G **do**
    $D[v] \leftarrow min\{D[v], D[w] + h(w,v)\}$

Algoritmus 2: Bellman–Fordov algoritmus

Časová zložitosť algoritmu je očividne $O(|V| \cdot |E|)$. Jeho korektnosť je možné nájsť napríklad v knihe [1, kapitola 3.3.4]. Toto dielo navyše obsahuje aj implementáciu spomínaného drobného vylepšenia, a to zisťovanie prítomnosti záporného cyklu.



Obr. 4: Orientovaný graf s ohodnotenými hranami

Funkcionalitu algoritmu si opäť predvedieme na jednoduchom príklade. Nech je vstupným grafom algoritmu graf vyobrazený na obrázku 4. Naším cieľom bude nájsť najkratšiu cestu z vrcholu 1 do ostatných vrcholov. Pod týmto popisom sa nachádza aj tabuľka (tabuľka 1) ukazujúca kľúčové hodnoty počas vykonávania algoritmu na našom príklade. Iterácia číslo 0 zaznamenáva stav vrcholov po inicializácií a pred začatím vykonávania cyklu, preto má počiatočný vrchol v tomto momente hodnotu $D[1] = 0$ a všetky ostatné $D[j] = \infty$. Počas iterácie $i = 1$ sa ideme pozerať na všetky hrany v grafe. Tu závisí od ich usporiadania, ktorá z nich sa vezme v akom poradí. V našom príklade, aby sme demonštrovali čo najviac možností, povedzme, že ako prvú určíme na porovnávanie hranu medzi vrcholmi 2 a 3. Keďže je ale hodnota $D[2]$ aj $D[3]$ rovná nekonečnu, zapíšeme do $D[3]$ opäť nekonečno. Nech je ďalšou hranou v poradí vyhodnocovania hrana medzi vrcholmi 1 a 2. Tu je už situácia príznačnejšia. Nakoľko poznáme hodnotu $D[1]$, môžeme zaznačiť, že $D[2] = 9$. Pozorné oko si isto všimne, čo by sa stalo, ak by boli vyhodnocovania týchto dvoch hrán v opačnom poradí. Totiž, v takom prípade by sme ihneď vedeli vypočítať i hodnotu $D[3]$. Pokračujme hranou medzi vrcholmi 1 a 4. Zisťujeme, že $D[4] = 10$. Hrana medzi vrcholmi 3 a 4 dovŕši prvú iteráciu algoritmu. Tu sa ale nič nezmení, keďže porovnávame nekonečno s hodnotou 10. V nasledujúcej iterácií $i = 2$ sa pre hrany spájajúce vrcholy $1, 2$ a $1, 4$ nestane nič zaujímavé. Pozrime sa teda na zvyšné hrany. Pre hranu medzi vrcholmi 2 a 3 sa upraví hodnota $D[3]$ na 12 a následne sa pre poslednú hranu (medzi vrcholmi 3 a 4) vykoná porovnanie hodnôt $(D[4] = 10)$ a $((D[3] = 12) + (-4))$. Po druhej iterácií sme už dospeli k optimálnemu riešeniu, čo ale náš algoritmus nemôže vedieť, a preto vykoná i poslednú, tretiu iteráciu. V nej sa ale už nič nezmení. Ak v dvoch stavoch po sebe nastala rovnaká situácia, respektíve neprišlo k žiadnej zmene, je jasné, že algoritmus dosiahol výsledok. Ak by teda bola na pláne ešte ďalšia iterácia či nebodaj viac, môžeme vykonávanie programu ukončiť bez strachu pre nesprávny výsledok. Mnohé implementácie sú preto väčšinou doplnené o toto jemné vylepšenie. V našom príklade by nám to však nepomohlo, keďže by program tak či tak zastavil po tretej iterácií.

Tabuľka 1: Priebeh algoritmu

iterácia = i	D[1]	D[2]	D[3]	D[4]
0	0	∞	∞	∞
1	0	9	∞	10
2	0	9	12	8
3	0	9	12	8

V prípade Bellman–Fordovho algoritmu je nám opäť na obtiaž jeho strohá funkci-

onalita, a teda zisťovanie cien najlacnejších ciest, zatiaľ čo my žiadame poznať celú nájdenú cestu. Modifikácia kódu je ale jednoduchá, rovnaká ako pre Dijkstru - pre každý vrchol si navyše zapamätáme, z ktorého vrcholu sme sa doň dostali.

Bellman–Fordov algoritmus má mnoho pozitív - aplikovateľnosť i na hrany so záporným ohodnotením. Jeho časová zložitosť je taktiež potešujúca a implementácia prehľadná. Avšak jeho funkčnosť i nad zápornými ohodnoteniami nám nijako nepomôže pri jeho aplikácií na grafikon hromadnej dopravy. Navyše, jeho časová zložitosť $O(|V| \cdot |E|)$ bude pri riešení nášho problému iba príťažou, nakoľko predpokladáme, že sieť liniek MHD bude obsahovať veľké množstvo hrán. Netreba však zabúdať, že algoritmus je pri iných problémoch široko využiteľný a výborný pre dobre zvolené dátové štruktúry.

2.2.3 Floyd–Warshallov algoritmus

Ďalším algoritmom slúžiacim na hľadanie najlacnejších ciest, ktorý uvedieme, nesie názov Floyd–Warshallov algoritmus. Opäť, ako v predošlom prípade, sa ale ani Floyd, ani Warshall nemôžu pýšiť prvenstvom. Predbehol ich Roy publikáciou z roku 1959, zatiaľ čo Floydova i Warshallova boli obe z roku 1962. Warshall dokonca v jeho práci nadväzuje na Kleeneho algoritmus z roku 1956. Finálna verzia, ktorú aj my predvedieme, dostala svoju podobu až na konci roku 1962 zásluhou Ingermana. Vďaka mnohým publikáciám sa tento algoritmus nazýva rôznymi menami. My si ale vystačíme s jedným.

Výsledok Floyd–Warshallovho algoritmu sa značne odlišuje od predošlých dvoch. Zatiaľ čo tie spoľahlivo zistili hodnoty najlacnejších ciest z daného vrcholu do všetkých ostatných vrcholov, Floyd–Warshallov ich vyráta pre všetky dvojice vrcholov. Samozrejme, to by sme vedeli dosiahnuť aj použitím predošlých algoritmov na všetkých vrcholoch. Vyžadovalo by sa však mnoho réžie navyše a kód by sa pravdepodobne priveľmi zneprehľadnil. Naopak, Floyd–Warshallov algoritmus je na pohľad až príliš jednoduchý, no nesmierne účinný.

Nami uvedená implementácia bude pracovať s maticami. Na vstupe dostane algoritmus graf $G = (V, E)$ bez záporných cyklov a incidenčnú maticu $W = (w_{ij})$ rozmerov $|V| \times |V|$. Maticu W si vieme jednoducho zostrojiť z hodnotiacej funkcie takto: Nech sú ceny hrán hodnotené hodnotiacou funkciou $h : V \times V \to R$, kde, ako obvykle, $h(u, u) = 0$, $uv \notin E \implies h(u, v) = \infty$ a $h(u, v) \in O(1)$. Potom $w_{ij} = h(i, j)$ pre všetky i, j. Príklad takejto matice W je vyobrazený na 2.1. Výsledkom algoritmu je posledná matica, a tou je matica $C^{(|V|)}$.

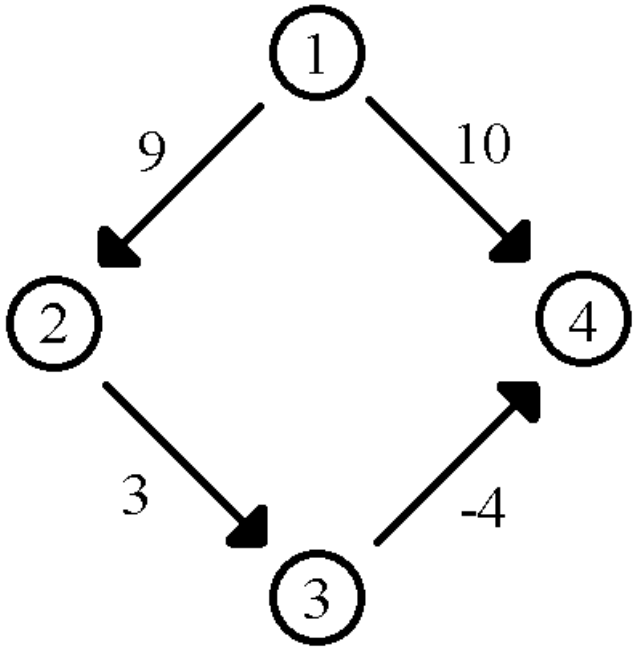
begin

$n \leftarrow |V|$
$C^{(0)} \leftarrow W$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $c_{ij}^{(k)} \leftarrow MIN(c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)})$
return $C^{(n)}$

end

Algoritmus 3: Floyd–Washallov algoritmus

Ľahko nahliadnuť, že časová zložitosť Floyd–Warshallovho algoritmu je $O(|V|^3)$. Jeho korektnosť je taktiež pohľadom badateľná. Pre istotu pridávame možnosť overiť si naše tvrdenie v knihe od Pavla Ďuriša, odkiaľ pochádza aj implementácia algoritmu [19, kapitola 2.2.2].



Obr. 5: Orientovaný graf s ohodnotenými hranami

Ako príklad nám poslúži ten istý graf ako v prípade Bellman–Fordovho algoritmu. Avšak výpočet algoritmu by bol na popis pridlhý, preto uvedieme iba zopár vyhodnocovaní a zaujímavé idey, ktoré je vhodné si uvedomiť. Prvý cyklus algoritmu, pracujúci s premennou k, vytvorí zakaždým celú novú maticu $C^{(k)}$. Pre každý riadok a stĺpec (premenné i a j) tejto novovznikajúcej matice sa vykoná porovnanie, ktoré vypočíta hodnotu momentálneho prvku. Napríklad, ak by sme chceli zrátať hodnotu v matici $C^{(1)}$ na pozícií $i = 2, j = 2$, s ľahkosťou využijeme vzorec použitý v algoritme, a teda $c^{(1)}_{2,2} = MIN(c^{(0)}_{2,2}, c^{(0)}_{2,1} + c^{(0)}_{1,2})$, po dosadení $c^{(1)}_{2,2} = MIN(0, \infty + 9) = 0$. Lepším príkladom môže byť zmena hodnoty v po sebe nasledujúcich maticiach, ako napríklad $c^{(3)}_{2,4} = MIN(c^{(2)}_{2,4}, c^{(2)}_{2,3} + c^{(2)}_{3,4}) = MIN(\infty, 3 + (-4)) = -1$.

$$C^{(0)} = W = \begin{pmatrix} 0 & 9 & \infty & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & -4 \\ \infty & \infty & \infty & 0 \end{pmatrix} \tag{2.1}$$

$$C^{(1)} = \begin{pmatrix} 0 & 9 & \infty & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & -4 \\ \infty & \infty & \infty & 0 \end{pmatrix} \tag{2.2}$$

$$C^{(2)} = \begin{pmatrix} 0 & 9 & 12 & 10 \\ \infty & 0 & 3 & \infty \\ \infty & \infty & 0 & -4 \\ \infty & \infty & \infty & 0 \end{pmatrix} \tag{2.3}$$

$$C^{(3)} = \begin{pmatrix} 0 & 9 & 12 & 8 \\ \infty & 0 & 3 & -1 \\ \infty & \infty & 0 & -4 \\ \infty & \infty & \infty & 0 \end{pmatrix} \tag{2.4}$$

$$C^{(4)} = \begin{pmatrix} 0 & 9 & 12 & 8 \\ \infty & 0 & 3 & -1 \\ \infty & \infty & 0 & -4 \\ \infty & \infty & \infty & 0 \end{pmatrix} \tag{2.5}$$

Opäť je namieste podotknúť, že na nami požadované vylepšenie na poznanie i predchodcov s cieľom rekonštrukcie nájdených najlacnejších ciest treba pridať ešte ďalších $|V|$ matíc, ktoré si danú informáciu budú počas výpočtu ukladať. Ich definíciu je možné uzrieť v už menovanom zdroji [19, kapitola 2.2.2].

Taktiež je vhodné poznamenať, že pre väčšie vstupy má algoritmus potenciál skonzumovať príliš veľa pamäte. Preto je v takom prípade výhodné algoritmus upraviť tak, aby používal iba dve matice, nakoľko pri algoritme je potrebné držať referenciu len na momentálnu a predchádzajúcu maticu ($C^{(k)}$ a $C^{(k-1)}$). To isté možno tvrdiť aj o maticiach predchodcov.

Floyd–Warshallov algoritmus je taktiež dobrým kandidátom pre riešenie nášho problému. Implementácia je opäť jednoduchá, časová zložitosť vítaná. Znovu ale, možnosť využitia algoritmu i na záporných hranách nie je pre nás nijako prínosná. Naviac, pri vyhľadávaní spojenia medzi dvomi zastávkami v grafikone MHD by bolo nutné vyrátať hodnoty najlacnejších ciest z každého vrcholu do všetkých vrcholov, čo môže trvať pridlho. Samozrejme, po skončení hľadania budú už nasledujúce dopyty uspokojené v konštantnom, resp. lineárnom (ak uvažujeme i rekonštrukcie ciest) čase.

2.2.4 Zhrnutie

Všetky tri prezentované algoritmy sú vhodnými adeptami pri riešení problému vyhľadávaní spojení v grafikone MHD. Každý z nich má svoje pozitíva, ale i nedostatky. Ktorý z algoritmov bude najvhodnejší, závisí hlavne od dát, nad ktorými bude pracovať. Ako

sme už spomínali, Bellman–Fordov algoritmus bude mať výhodu pri grafoch obsahujúcich menšie množstvo hrán, čo, žiaľ, nebude prípad liniek MHD. Floyd–Warshallov algoritmus je zasa zostavený tak, aby vypočítal všetky najlacnejšie cesty. Preto bude jeho priebeh pri väčších vstupoch neprimerane dlhý. Jeho využitie preto vidíme pri grafikonoch malých miest, kde sa vyrátajú všetky potrebné hodnoty už pri inicializácií, najlepšie na druhom vlákne, aby mohol používateľ zadávať dopyt už počas výpočtu. Využitie algoritmu by bolo v tomto prípade priam žiadané, keďže ostatné dva algoritmy môžu svoje výpočty pri dlhšej postupnosti dopytov opakovať, čo ich môže zbytočne zdržovať. A nakoniec, Dijkstrov algoritmus sa zdá byť najvšeobecnejší a najpoužiteľnejší pre ľubovoľné dáta. Neobsahuje žiadnu dodatočnú výhodu, ale takisto nemá ani žiadne nedostatky, ktoré by sme mu vedeli vytknúť.

Existujú aj iné, pokročilejšie algoritmy, ktoré sú schopné nájsť najlacnejšie cesty v grafe. Zvyčajne však ide o kombináciu nami uvedených algoritmov, prípadne ich optimalizácia alebo nejaká dômyselná modifikácia. Na predstavu poslúži Johnsonov algoritmus, ktorého výstup je rovnaký ako výsledok Floyd–Warshallovho algoritmu, len jeho priebeh je iný - využíva kombináciu Dijkstrovho a Bellman–Fordovho algoritmu, čím dosahuje na riedkych grafoch výsledok rýchlejšie. Takýmto algoritmom sa však v našej práci zaoberať nebudeme. Viac sa radšej zameriame na algoritmy prispôsobené na vyhľadávanie v grafikonoch.

2.2.5 Vylepšenia a rozšírenia

V nasledujúcom uvedieme niektoré vylepšenia doposiaľ popísaných algoritmov.

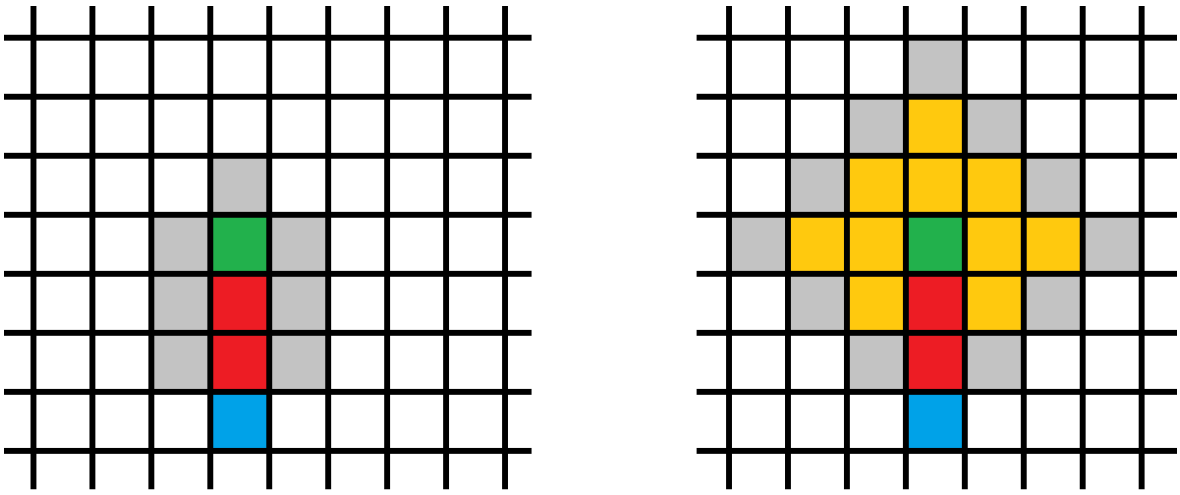
Algoritmus A*

Vyhľadávací algoritmus A* je založený na princípe Dijkstrovho algoritmu. Prišiel s ním Nils Nilsson v roku 1964, v roku 1967 ho Peter E. Hart vylepšil a o rok neskôr dokázal Bertram Raphael jeho optimálnosť, pričom zaviedol i pomenovanie A* (alebo aj *A star*).

Algoritmus dokáže vyhľadávať ceny najlacnejších ciest iba medzi počiatočným a koncovým vrcholom. Navyše, algoritmus pracuje s heuristickou funkciou, čo môže zapríčiniť jemné zhoršenie v optimálnosti nájdených riešení.

Ako sme už spomínali, algoritmus sa veľmi podobá Dijkstrovmu. Rozdiel je ten, že pri vyberaní nasledujúceho vrcholu pre nadchádzajúcu iteráciu algoritmu berieme vrchol s najnižšou hodnotou $f(v)$, kde v sú všetky relevantné vrcholy pre ďalšiu iterá-

ciu. Funkcia $f(x) = g(x) + h(x)$ pre nejaký vrchol x, pričom funkcia $g(x)$ predstavuje cenu nájdenej najlacnejšej cesty od začiatočného vrcholu do vrcholu x a funkcia $h(x)$ odhaduje cenu najlacnejšej cesty z vrcholu x do koncového vrcholu. Funkcia $h(x)$ teda predstavuje spomínanú heuristickú funkciu. Navyše, táto heuristická funkcia nemôže nadhodnotiť cenu najlacnejšej cesty k cieľu, inak by algoritmus mohol prehliadnuť najoptimálnejšie riešenie. Táto vlastnosť sa nazýva *prípustnosť*. Na porovnanie, ak uvažujeme graf, kde sú všetky hrany ohodnotené rovnako, Dijkstrov algoritmus bude prehľadávať graf do všetkých strán rovnomerne, zatiaľ čo A* algoritmus bude, vďaka heuristike, vyberať hrany, ktoré dospejú skorej ku koncovému vrcholu. Ako je vyobrazené na 6, Dijkstrov algoritmus sa rozpína na všetky strany a algoritmus A* si ihneď vyberá vrchol najbližšie k cieľovému. Zelenou farbou je znázornený počiatočný vrchol, modrou koncový. Žltou farbou sú vyobrazené algoritmom prejdené vrcholy, šedou tie naplánované a červenou nájdená cesta.



Obr. 6: Porovnanie A* (vľavo) a Dijkstrovho algoritmu (vpravo) na mriežke

Pri algoritme A* môže byť problémové stanoviť heuristickú funkciu. Zjavne, ak $h(x) = 0$ pre všetky vrcholy x, výber vrchola pre iteráciu algoritmu bude závisieť len od doposiaľ zistenej ceny cesty $g(x)$, a teda bude prebiehať rovnako ako Dijkstrov algoritmus. Pre každú inú heuristickú funkciu, ktorá si ale zachová prípustnosť, je garantované zlepšenie v rýchlosti výpočtu algoritmu a aj zachovanie optimálnosti. Jednou z možností, ako nájsť dobrú heuristickú funkciu, je zobrať *euklidovskú vzdialenosť* (vzdialenosť vzdušnou čiarou) medzi uvažovaným a koncovým vrcholom. V našom prípade uvažujeme čas prepravenia sa medzi dvoma zastávkami, takže budeme musieť počítať i s rýchlosťou najrýchlejšieho spojenia, a tak zistili teoreticky najlepší čas cesty medzi nimi.

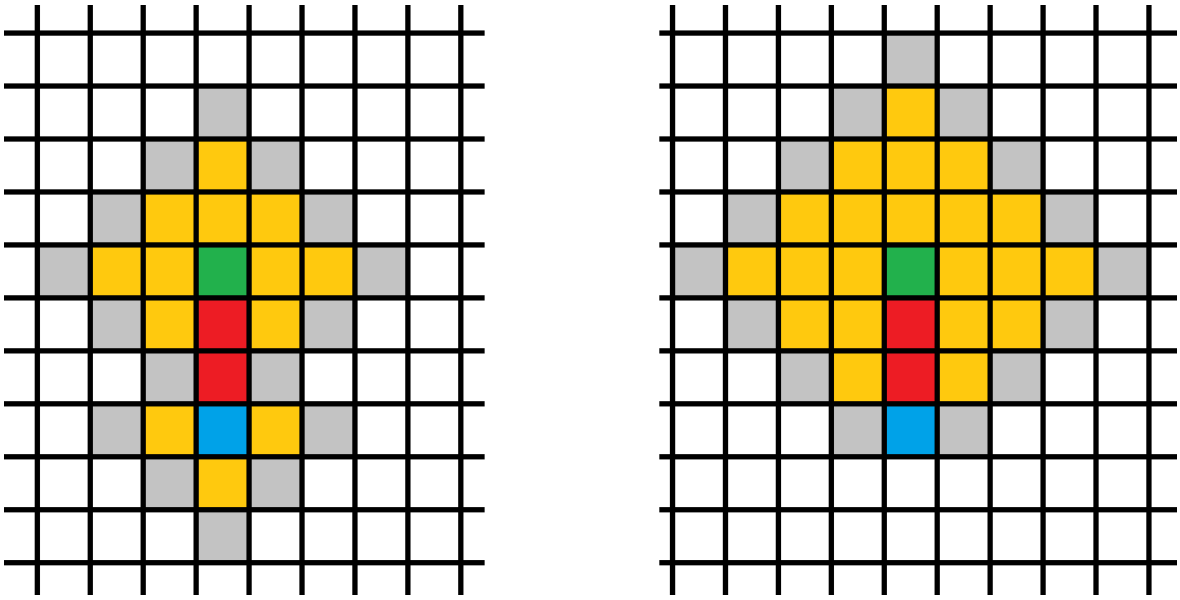
Časová zložitosť algoritmu je $O(|E|)$. Alebo inak, predstavme si výsledok vyhľadávania ako strom. Označme b faktor vetvenia vzniknutého stromu, teda priemerný počet hrán vychádzajúci z vrchola a nech d je jeho hĺbka. Potom časová zložitosť algoritmu je $O(b^d)$. S týmito úvahami môžeme doplniť i informáciu o časovej zložitosti Dijkstrovho algoritmu, ktorá je teda tiež $O(b^d)$.

Obojstranné vyhľadávanie

Myšlienka obojsmerného vyhľadávania je jednoduchá. Napriek tomu s jej aplikáciou na Dijkstrovom algoritme prišiel až Ira Pohl v roku 1971. A s jej použitím na algoritme A* prišiel v roku 1994 Tetsuomi Ikeda.

Priebeh algoritmu bude teda nasledovný: Počiatočný a koncový vrchol, ktoré sú vstupnými údajmi algoritmu, sa spracujú a spustí sa na nich samotný vyhľadávací algoritmus dva krát. Prvý bude vyhľadávať od počiatočného, druhý bude spätne vyhľadávať od koncového vrcholu. Pri spätnom vyhľadávaní na orientovanom grafe je nutné brať v úvahu opačné smery hrán, lepšie povedané, ak xy je hrana z x do y, vyhľadávanie pobeží na pomyselnej hrane yx. Koniec algoritmu nastane, ak sa niektoré z dvoch hľadaní dostane do vrcholu, ktorý už opačné vyhľadávanie spracovalo.

Je zjavné, že popísaná myšlienka je aplikovateľná na obi dva algoritmy –Dijkstrov i A*. Navyše, časová zložitosť je vo väčšine prípadov oveľa priaznivejšia. Nech b je faktor vetvenia a d hĺbka vzniknutého stromu, potom časová zložitosť oboch algoritmov bez zmeny je $O(b^d)$, avšak časová zložitosť s implementovaným obojsmerným prehľadávaním je $O(2.(b^{d/2}))$. Zjavne, ak je faktor vetvenia $b = 2$, nejde o nijaké zlepšenie. Avšak takáto situácia nastáva len pri naozaj riedkych grafoch. Pri iných je hodnota b oveľa väčšia, preto je poskytnuté zlepšenie naozaj výhodné. Počet algoritmom prejdených vrcholov býva v priemere približne polovičný.



Obr. 7: Porovnanie obojsmerného Dijkstrovho algoritmu (vľavo) a jednosmerného (vpravo) na mriežke

Zhrnutie

Obi dve vylepšenia prinášajú zefektívnenie vyhľadávania oproti základným algoritmom, ktoré sme poskytli v predošlej sekcií. Avšak naše myšlienky smerujú k vytvoreniu aplikácie, ktorá nebude využívať pripojenie k internetu, a teda nebude vedieť zistiť vzdialenosť vzdušnou čiarou medzi dvomi zastávkami. Algoritmus A* preto nebude pre nás vhodnou voľbou. A zas, pridanie obojstranného vyhľadávania by iba jemne zrýchlilo aplikáciu za cenu znáročnenia a zneprehľadnenia kódu. My sa preto uspokojíme so schopnosťami Dijkstrovho algoritmu.

Uvedené vylepšenia sú iba názorným príkladom vybraných algoritmov z pestrej škály rôznych vylepšení a následných modifikácií základných algoritmov určených na vyhľadávanie v grafikonoch. Existujú teda i rýchlejšie spôsoby zistenia najlacnejších ciest v grafoch, a to prevažne pomocou predpočítania si rôznych hodnôt a údajov. Dobrým príkladom môže byť takzvaný *Highway-node routing*[18] alebo jeden z jeho špeciálnych prípadov –*Contraction hierarchies*[10]. Hlavnou myšlienkou týchto úvah je vytvorenie hierarchie z liniek tak, že najefektívnejšie v zmysle dopravenia sa do cieľa budú v jednej skupine, menej efektívne v ďalšej, a tak až po okrajové linky mesta. Algoritmus sa potom snaží vyberať linky s čo najväčšou efektívnosťou.

Viaceré prístupy a spôsoby premýšľania nad problematikou vyhľadávania ciest zhŕňa dielo z roku 2010 [3]. Množstvo výskumov, experimentov či porovnaní existujú-

cich implementácií uvádzajú vo svojom diele Fan a Machemehl [9].

2.2.6 Existujúce implementácie vyhľadávacích algoritmov

V dnešnej dobe existujú mnohé vyhľadávače spojení využívajúce popísané algoritmy. Samozrejme, ich implementácie sa môžu značne odlišovať od tých nami poskytnutých, nakoľko je potrebné zabezpečiť rôzne dodatočné informácie, možnosti cesty či spôsoby vyhľadávania.

Dobrým príklad preto môže byť program *graphhopper*[1], ktorý implementuje Dijkstrov algoritmus, algoritmus A* a dokonca aj ich obojstranné vyhľadávania.

Zaujímavým môže byť aj vyhľadávač spojení *Explorer* [20]. Ten implementuje Dijkstrov algoritmus, avšak využíva mnohé predpočítania aby zefektívnil svoje výpočty.

Možnosť vyhľadávania a plánovania ciest poskytuje aj spoločnosť *Google*. Samozrejme, neexistujú oficiálne vyjadrenia, ale podľa mnohých domnienok[2] používa ich implementácia na zistenie najkratších ciest spôsob *Contraction hierarchies* s množstvom predpočítavania a iných úprav. Ten istý prístup používa i projekt *OSRM*[3].

Iným pohľadom na problematiku je algoritmus *RAPTOR*[5], ktorý nepoužíva princíp Dijkstrovho algoritmu. Na jeho základe je postavený program *navitia*[4].

[1]dostupný na: https://github.com/graphhopper/graphhopper

[2]Napríkald https://stackoverflow.com/questions/14091279/algorithm-method-use-by-google-maps-for-finding-the-path-between-two-cities

[3]Dostupný na: http://project-osrm.org/

[4]Dostupný na: https://github.com/CanalTP/navitia

Kapitola 3

Softvér

Táto kapitola obsahuje základné informácie o nami implementovanom softvéri. Popisujeme v nej, ako program funguje, na čo slúži a ako sa ovláda. Uvádzame rôzne funkcie softvéru, prípadne aj príklady ich použitia. Implementácia softvéru v prostredí Unity je dostupná na stránke "https://github.com/LordLoles/MHD-Bakalarka".

3.1 Základný popis aplikácie

Aplikácia je vytvorená tak, aby používateľ intuitívne vedel, čo na čo slúži.

Obr. 8: Ukážka spusteného programu

Na obrázku 8, možno vidieť textové polia pod textami *Začiatok* a *Koniec*, ktoré použijú zadaný text ako začiatočný a konečný bod pre vyhľadávací algoritmus. Upozorňujeme, že názvy zastávok musia byť zadané presne tak, ako sú uložené v internej databáze, teda v textovom súbore, ktorý neskôr popíšeme.

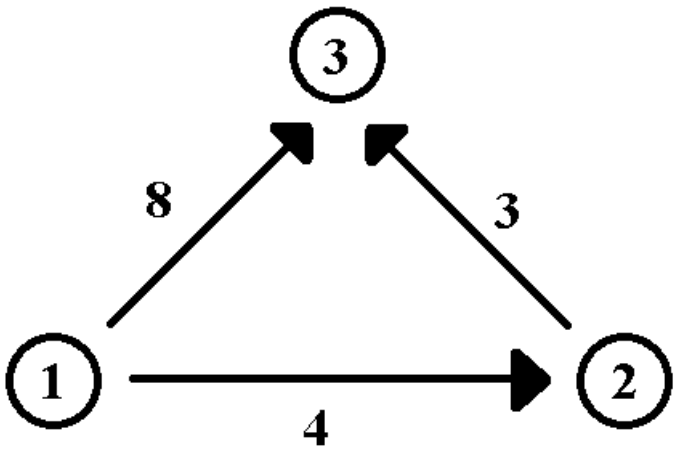
Textové pole *Čas* berie za vstup, ako je v poli naznačené, čas vo formáte "hh:mm". Ak je prvá hodnota nula, netreba ju zadávať. Správne vstupy sú napríklad "5:32", "17:08"či "9:2". Ak bude formát vstupu zlý, prípadne používateľ nechá pole prázdne, vyhľadávanie začne v momentálnom čase, aký je nastavený na používanom zariadení.

Pole *Počet* požaduje vyplniť číslo, ktoré určí, koľko výsledkov sa má najviac zobraziť. Opäť, pri nesprávnom vstupe či nevyplnenom poli sa použije hodnota 3.

Tlačidlo *Hľadaj* spracuje vstupné údaje a spustí s nimi vyhľadávanie. Ak sa stále ešte načítavajú údaje z databázy, nie je možné vyhľadávanie spustiť. Počas načítavania sa ale dajú vpisovať požadované parametre do textových polí.

3.2 Vyhľadávanie

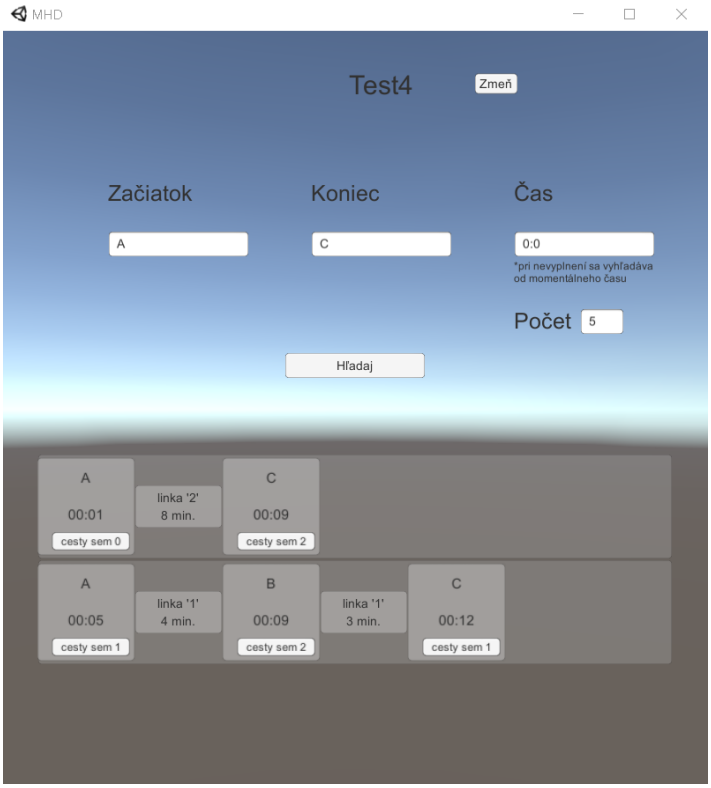
Funkčnosť vyhľadávania ukážeme na jednom z našich testových vstupov. Nech je vstupný grafikon rovnaký ako na obrázku 9 a uvažujme linku '1' vyrážajúcu z A cez B do C v časoch 0:1 a 0:5 a linku '2' v čase 0:1 premávajúci iba z A do C.



Obr. 9: Príklad testového vstupu

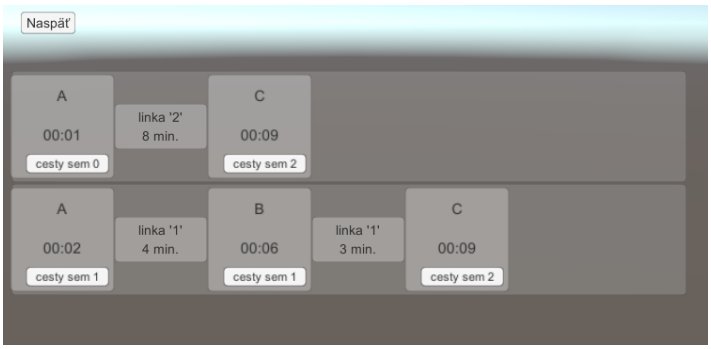
Spustime teda vyhľadávanie z A do C v čase 0:0. Vidieť, že sa do požadovanej

zastávky môžeme dostať v časoch 0:9 a 0:12. Možno si povšimnúť, že linka '1' i linka '2' prídu obe v čase 0:9 do konečnej zastávky.



Obr. 10: Ukážka vyhľadávania

Vidíme, že výsledok vyhľadávania aplikácie je správny. Navyše, pri zastávke *C* v čase 0:9 je tlačidlo *cesty sem 2*, ktoré naznačuje, že existujú, ako sme i my usúdili, dve relevantné cesty do tejto zastávky v tom istom čase. Po kliknutí na tlačidlo sa obe zobrazia spolu s tlačidlom *Naspäť*, ktoré zobrazí predošlé výsledky.



Obr. 11: Ukážka alternatívnej cesty do zastávky C

3.3 Dátové súbory

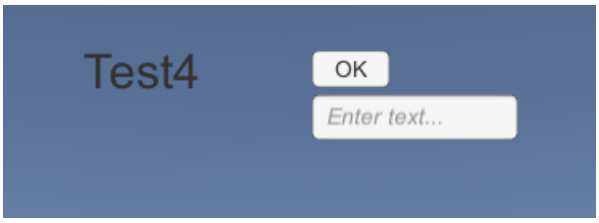
Naša aplikácia je konštruovaná tak, aby si každý jej požívateľ mohol vytvoriť vlastné dáta, s ktorými má program pracovať.

Zložka so spustiteľným súborom aplikácie obsahuje i zložku *build_Data* a tá zložku *Data*. V nej sa nachádzajú rôzne testové vstupy. Stačí teda vytvoriť zložku s príznačným názvom a do nej umiestniť dva textové súbory s názvami *zastavky* a *linky*. Súbor *zastavky* má obsahovať názvy všetkých zastávok. Súbor *linky* je trochu zložitejší. Ten obsahuje linky, pričom každú popisuje 5 riadkov súboru. V prvom je názov linky. Zvyšné štyri sú rozdelené - prvé dva značia cestu linky tam, druhé dva cestu linky naspäť. Tieto dva a dva riadky majú rovnaký formát, preto z nich popíšeme len jednu dvojicu. V prvom riadku z dvoch sa nachádza vždy čas od počiatočnej zastávky sem a názov zastávky v chronologickom poradí. Táto dvojica času a názvu je od ďalšej oddelená znakom | . Druhý riadok obsahuje časy, v ktorých linka vyráža z počiatočnej zastávky. Formát tohto súboru sa môže javiť mierne chaotický, preto uvádzame i obrázok 12, ktorý ho názorne popisuje.

```
"Nazov linky"
"cas cesty od vyrazenia z pociatocnej zastavky" "Nazov zastavky" | ...                    //cesta tam
"cas vyrazenia z pociatocnej zastavky" "cas vyrazenia z pociatocnej zastavky" ...        //cesta tam
"cas cesty od vyrazenia z pociatocnej zastavky" "Nazov zastavky" | ...                   //cesta spat
"cas vyrazenia z pociatocnej zastavky" "cas vyrazenia z pociatocnej zastavky" ...       //cesta spat
"Nazov linky"
...


PRIKLAD
1
0 Stanica | 2 Ulica A | 5 Namestie C
8:52 12:34 16:10 21:53
0 Namestie C | 3 Ulica A | 5 Stanica
9:22 12:50 16:31 21:59
```

Obr. 12: Formát dátového súboru pre linky

Po vytvorení zložky s dátami už stačí len v programe kliknúť na tlačidlo *Zmeň*, čo zobrazí textové pole, do ktorého používateľ zadá názov vytvorenej zložky a klikne na to isté tlačidlo (teraz už s nápisom *OK*). Program sa následne postará o načítanie dátových súborov, čo zapríčiní i zmenu textu hneď vedľa spomínaného tlačidla na názov poskytnutej zložky.



Obr. 13: Zmena dát, s ktorými má program pracovať

Kapitola 4

Implementácia softvéru

V tejto kapitole uvedieme, ako sme postupovali pri implementácií softvéru. Odhalíme, aké postupy, štruktúry či algoritmy sme zvolili, pričom popíšeme aj dôvody nášho výberu. Neuvádzame celý kód, ale len tie časti, ktoré sa nám zdali byť najviac zaujímavé a zodpovedajúce obsahu našej práce.

4.1 Prvé myšlienky

V prvom rade je zo všetkého najdôležitejšie ujasniť si, aké ciele má spĺňať naša implementácia. Treba si uvedomiť, čo chceme, aby náš program vedel robiť, jemne si načrtnúť, ako potrebné veci naprogramovať, ale tiež v akom poradí na nich pracovať.

Prvoradý a najpodstatnejší cieľ našej práce je dozaista vyhľadávač spojení v grafikone mestskej hromadnej dopravy. Teraz rozoberieme, čo všetko jeho implementácia obnáša.

Na začiatok budeme potrebovať vytvoriť, prípadne získať dáta, s ktorými budeme pracovať. Tie môžeme buď náhodne vygenerovať, pre vlastnú potrebu a testovanie aplikácie prevziať dáta z už existujúcej siete MHD, alebo si vytvoriť nejaké vlastné, sofistikované testovacie vstupy. Každá možnosť má svoje kladné i záporné stránky. Náhodné dáta otestujú algoritmy veľmi slušne, avšak môžu obsahovať rôzne anomálie a pravdepodobne ani zďaleka nebudú zobrazovať reálny tvar grafikonu MHD. Prevzaté dáta z existujúcich MHD tieto nedostatky znamenite riešia. Ich nevýhodou je ale ich použitie len na lokálne účely, nakoľko nemôžu byť publikované verejne. Vlastne vytvorené vstupy sú zasa výborné na testovanie okrajových prípadov, na nachádzanie a odstraňovanie chýb v programe, ako i na skúšanie programu pri malých zmenách v algoritme a pozorovaní očakávaných výsledkov. Naopak ale, postrádajú prítomnosť

rozmerných dátových súborov, keďže ručné vytváranie objemných dát je časovo priveľmi náročné.

V našich mysliach teda leží otázka, aké dáta zvoliť ako vstup. Samozrejme najlepšie by bolo použiť všetky tri spomínané možnosti. My sa však uspokojíme iba s dvomi z nich. Pri vytváraní kódu bude veľmi vhodné využívať nami navrhnuté vstupy, ktoré otestujú práve vytváranú časť programu. Po vytvorení zdanlivo funkčnej aplikácie ju otestujeme na existujúcich dátach, pričom zistíme či je program dostatočne rýchly a či sú jeho výstupy uspokojivé v porovnaní s inými vyhľadávačmi spojení. Náhodne generované vstupy nebudeme používať, keďže pomocou kombinácie predošlých dvoch zistíme všetky potrebné informácie o funkčnosti aplikácie.

Treba si tiež premyslieť, kde si budeme spomínané dáta uschovávať. Za uváženie stoja dve možnosti. Dáta držať v textovom súbore alebo ich načítať do nejakej databázy, a z nej potom čerpať. Textová reprezentácia je všestranná, ľahko s ňou pracovať i meniť vstupné údaje. Jej načítanie môže ale spomaľovať chod našej aplikácie. Naproti tomu, databáza je rýchla na čítanie z nej, ale jej o niečo ťažšie vytváranie znepríjemňuje našu snahu o testovanie mnohých vlastných vstupov. My sme, nakoľko je to jednoduchšie, zvolili textovú reprezentáciu dát. Opäť, najvhodnejšie by bolo oba spôsoby skombinovať, a teda uschovávať údaje v textových súboroch a nejaký vybraný vstup mať uložený v databáze.

Ďalším bodom, o ktorom radno pouvažovať, je reprezentácia týchto údajov. Pre začiatok volíme znova čo najjednoduchšiu možnosť: vrcholmi nami používaného grafu budú zastávky MHD a hranami sa stanú linky jazdiace medzi nimi. Tu, keďže sme si vybrali takú jednoduchú alternatívu, nesmieme zabúdať na flexibilitu. Ak sa ukáže nejaký problém, netreba sa báť reprezentáciu údajov rozumne pozmeniť, aby sme si zbytočne nenarobili problémy, ktoré sa budú v neskorších častiach implementácie náročne odstraňovať.

Nasledujúcou zastávkou je vytvorenie objektov na reprezentáciu grafu. Už sme spomínali, že budeme pracovať s orientovanými grafmi. Sprvu si vystačíme s hodnotením hrán. Teraz presnejšie popíšeme objekty, ktoré si budú držať informácie o vrcholoch, hranách i o celom grafe a budú tak reprezentovať uvedené grafové štruktúry. Riešenie je nateraz triviálne: Trieda vrcholu moc informácií nepotrebuje. Nech je jej premennou len názov zastávky. Trieda hrany bude vyžadovať referencie na dva vrcholy, keďže sme tak hrany v kapitole 1 definovali, a premennú predstavujúcu hodnotiacu funkciu. Táto implementácia sa jemne líši od našej definície hodnotiacej funkcie, avšak uvedeným spôsobom poľahky dosiahneme všetky predpoklady spomínanej funkcie. Ak poznáme

hranu, v konštantnom čase vieme zistiť jej hodnotenie, keďže len pristúpime k zodpovedajúcej premennej. Nakoniec, trieda grafu bude obsahovať dve polia reprezentujúce všetky hrany a všetky vrcholy v grafe.

V tejto chvíli je prvotne premyslená celá reprezentácia údajov a prichádza na rad výber vyhľadávacieho algoritmu. Rozdiely medzi uvedenými algoritmami sa nachádzajú na konci predošlej kapitoly (2.2.4). Z nich vychádzajúc sme usúdili, že najvhodnejšie bude implementovať Dijkstrov algoritmus. Naše úvahy vychádzali prevažne z nami vytýčených cieľov, a teda funkcionality programu na reálnych dátach, ktoré sú vo väčšine prípadov veľké zoskupenia údajov. Z tohto dôvodu je nevhodný ako Bellman–Fordov, tak i Floyd–Warshallov algoritmus. Dijkstra navyše spoľahlivo a rýchlo funguje aj na menších vstupoch a pre jeho všestrannosť je najlepším kandidátom pre riešenie našich cieľov.

V našom hypotetickom modeli už teda dokážeme zistiť výsledok vyhľadávania. Avšak musíme ešte ošetriť, kam sa želaný stav uloží, aby bolo jednoduché ho vypísať. A taktiež si treba rozmyslieť, ako chceme výsledky vypisovať. Ako sme už spomínali v kapitole 2, budeme používať implementáciu vyhľadávacieho algoritmu s pamätaním si predchádzajúcich vrcholov pre nájdené najlacnejšie cesty, aby sme boli schopný ich zrekonštruovať. Mohli by sme teda vytvoriť nové pole o veľkosti počtu vrcholov grafu, do ktorého si tieto referencie uložíme. My však použijeme rovnaký postup ako pri hodnotiacej funkcií, a teda do objektu vrchola pridáme premennú držiacu si referenciu na predchádzajúci vrchol. Toto riešenie ale nie je najšťastnejšie, keďže objekt vrchola nemá čo do činenia s výsledkom vyhľadávania najlacnejších ciest v grafe. Správne by bolo všetky vrcholy obaliť ďalším objektom s príznačným názvom a v ňom si držať túto informáciu. V našom prípade ale ide o celkovo nie až tak ťažký kód, preto je takéto jemné zneprehľadnenie akceptovateľné. Vďaka memorizácií predchodcu vrchola nám teda stačí pri výpise poznať cieľovú zastávku vyhľadávania a dokážeme poľahky vyhotoviť žiadanú cestu. Pre začiatok sa uspokojíme s výpisom do konzoly, čo nám postačí až do záverečných prác na aplikácií.

Zostávajúcemu cieľu, čo je implementácia vylepšenia, sme sa rozhodli venovať a premýšľať nad ním až po uskutočnení doteraz opisovaného cieľa.

Po prejdení si všetkých potrebných úvah nám ale stále zostáva jedna nezodpovedaná otázka. Musíme si vybrať programovací jazyk, prípadne prostredie, v ktorom naše myšlienky zrealizujeme. Znalosť potrebných vecí na implementáciu nám ale výrazne pomôže pri našom výbere. Nakoľko budeme chcieť reprezentovať graf, jeho vrcholy i hrany, bude nanajvýš priaznivé zvoliť objektovo orientovaný jazyk. Na myseľ prichádza Java,

C#, Python, JavaScript. Ďalším kritériom je, že v aplikácií budeme od používateľa požadovať vstupné parametre v podobe počiatočnej a koncovej zastávky. Budeme teda vytvárať nejaké okná, textové polia a tlačidlá. Tieto možnosti sú už predpripravené, spolu s mnohými inými, vo vývojom prostredí Unity. Ide o prostredie určené prevažne na vývoj počítačových hier, avšak pre naše účely je úplne dokonalé. Využijeme jeho možnosť programovania v jazyku C# a hojne i jeho grafické rozhranie.

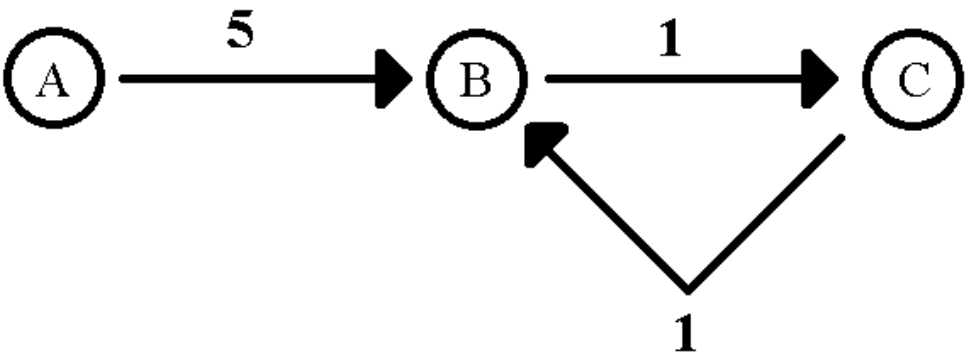
4.2 Začiatočná implementácia

Po dôkladnom premyslení si všetkých potrebných detailov sme prešli k prvej implementácií. Samozrejme, nevyriešili sme ich všetky. Hneď, ako sme chceli vytvoriť prvý testový vstup, zistili sme, že sme si neujasnili, v akom formáte budeme dáta uschovávať. Ich formát je nepodstatný pre beh aplikácie, keďže si dáta v programe ihneď spracujeme na nami navrhnuté štruktúry. Stačí teda zvoliť ľubovoľný rozumný. Náš je demonštrovaný na obrázku 14.

```
"Nazov linky"
"cas cesty od vyrazenia z pociatocnej zastavky" "Nazov zastavky" | ...                          //cesta tam
"cas vyrazenia z pociatocnej zastavky" "cas vyrazenia z pociatocnej zastavky" ...              //cesta tam
"cas cesty od vyrazenia z pociatocnej zastavky" "Nazov zastavky" | ...                         //cesta spat
"cas vyrazenia z pociatocnej zastavky" "cas vyrazenia z pociatocnej zastavky" ...             //cesta spat
"Nazov linky"
...


PRIKLAD
1
0 Stanica | 2 Ulica A | 5 Namestie C
8:52 12:34 16:10 21:53
0 Namestie C | 3 Ulica A | 5 Stanica
9:22 12:50 16:31 21:59
```

Obr. 14: Formát obsahu dátového súboru

Teraz môžeme smelo zbierať dáta. Vďaka stránke Bratislavského dopravného podniku sme si uložili dáta popisujúce mestskú hromadnú dopravu v Bratislave. Tie použijeme lokálne na testovanie aplikácie. Vytvorili sme aj pár vlastných vstupov. Na obrázku 15 uvádzame ich jednoduchý príklad. V ňom sme uvažovali o linke vyrážajúcej z A do C v časoch 0:2 a 0:5 so spiatočnou cestou iba z C do B v čase 0:1.



Obr. 15: Príklad testového vstupu

Nasleduje implementácia grafových štruktúr, ktorá je nateraz jednoduchá. Vhodné je poznamenať, že v našej implementácií vrchola, ktorú zachytáva obrázok 16 , si budeme pamätať hodnotu najlacnejšej cesty do daného vrcholu, aby sme v Dijkstrovom algoritme nepotrebovali vytvárať pole čísel uchovávajúce jeho výsledky. Taktiež si budeme pamätať i referenciu na hranu, ktorá tento vrchol spája s jeho predchodcom, čo nám uľahčí budúcu prácu.

```
public class Vertex
{
    internal string name;

    internal int value; //for dijkstra's purpose
    internal Vertex parent; //for dijkstra's purpose
    internal Edge toParent; //for dijkstra's purpose


    public Vertex(string n)
    {
        name = n;
    }
}
```

Obr. 16: Prvá reprezentácia vrcholu

Kód pre objekt hrany má ešte navyše pridané časy príchodu a odchodu linky z daných vrcholov. Inak objekty reprezentujúce hranu a graf neobsahujú žiadne zmeny.

```
public class Edge
{
    internal string name;
    internal Vertex fromV;
    internal Vertex toV;
    internal Time fromT;
    internal Time toT;
    internal int travellTime; //minutes


    public Edge(string name, Vertex fromV, Vertex toV, Time fromT, Time toT)
    {
        this.name = name;
        this.fromV = fromV;
        this.toV = toV;
        this.fromT = fromT;
        this.toT = toT;
        this.travellTime = Time.differenceBetweenTimesMin(this.fromT, this.toT);
    }
}
```

Obr. 17: Prvá reprezentácia hrany

```
public class Graph
{
    internal List<Vertex> verteces;
    internal List<Edge> edges;


    internal Graph()
    {
        verteces = new List<Vertex>();
        edges = new List<Edge>();
    }
}
```

Obr. 18: Prvá reprezentácia grafu

Pri vytváraní kódu bolo tiež potrebné rozobrať súbor s dátami a vytvoriť z nich horeuvedené objekty. Tento kód uvádzať nebudeme, nakoľko je dosť obsiahly a navyše neobsahuje žiadne relevantné informácie.

Ďalším krokom je implementácia vyhľadávacieho algoritmu. Tu však badáme komplikáciu. Medzi dvomi vrcholmi/zastávkami sa nachádza mnoho hrán reprezentujúcich všetky spojenia medzi nimi. Treba pomocou algoritmu zistiť, ktorú z týchto hrán vybrať. Najlepšie by bolo nejako vyberať hrany čo najskôr po udanom čase, od ktorého chceme vyhľadávať. To by však potrebovalo premnoho modifikácií algoritmu. My sme sa preto rozhodli riešiť tento problém zmenou reprezentácie údajov.

4.3 Vylepšenie reprezentácie údajov

Keďže nám predošlá reprezentácia údajov spôsobovala komplikácie pri vyhľadávaní, rozhodli sme sa ju prerobiť. Nejde o priveľmi signifikantnú zmenu, avšak jej dopad na vyhľadávanie bude nanajvýš uspokojujúci. Zmenou je, že naše vrcholy nebudú reprezentovať len zastávku, ale aj čas, v ktorom sa niečo v tej zastávke deje. Napríklad pre linku idúcu zo zastávky A do B v čase 0:1 s príchodom 0:6 vytvoríme dva vrcholy, jeden s hodnotami A a 0:1 a druhý s údajmi B a 0:6, pričom uvažovaná hrana spája presne tieto dva vrcholy. Tým pádom budeme nútení vytvoriť veľa hrán navyše, a to hrany, na ktorých sa čaká na nejakej zástavke. My sme tieto hrany nazvali "čakacie". Napríklad ak by bol v poslednom príklade práve čas $0:0$, prvá hrana nášho cestovania medzi A a B by bola hrana spájajúca vrcholy A v čase 0:0 a A v čase 0:1. Uvedené rozšírenie možno vidieť na obrázku 19.

```
public class Vertex
{
    internal string name;
    internal Time time;

    internal int value; //for dijkstra's purpose
    internal Vertex parent; //for dijkstra's purpose
    internal Edge toParent; //for dijkstra's purpose


    public Vertex(string n, Time t)
    {
        name = n;
        time = t;
    }
}
```

Obr. 19: Rozšírená reprezentácia vrchola

Do objektu hrany sme zasiahli len minimálne: pridali sme jednu premennú hovoriacu či je daná hrana čakacia, čo nám pomôže pri výpise cesty, pretože z neho budeme chcieť takéto hrany vylúčiť.

Táto reprezentácia údajov vyzerá naozaj sľubne, keďže algoritmus vyhľadávania na takto zostavenom grafe nebude potrebovať žiadne ďalšie úpravy.

4.4 Implementácia vyhľadávacieho algoritmu a výpis výsledku

Nasledujúcim bodom bolo teda implementovať vyhľadávací algoritmus na pripravenej grafovej štruktúre. Ako sme už spomínali, na tento účel využijeme Dijkstrov algoritmus. Jeho implementácia podľa podkapitoly 2.2.1 je nenáročná. Za zmienku ale stojí pýtať sa, ktorý vrchol, po zadaní požiadavky používateľom, zvolíme ako počiatočný. Odpoveď je ale prostá: nájdeme vrchol s daným názvom, ktorý je čo najskôr po zadanom čase používateľa. Druhou možnosťou by bolo vytvoriť nový vrchol s parametrami od používateľa a ten umiestniť do grafu. Implementácia by bola jednoduchšia, ale oveľa viac mätúca. Rozhodli sme sa preto pre prvú možnosť.

Opäť je vhodná chvíľa uvažovať. Ak bude náš graf rozsiahly, s čím musíme rátať, Dijkstrov algoritmus bude počítať veľmi dlho, než sa dopátra ku všetkým hodnotám. Rozumné je teda do implementácie zapojiť i ukončenie procesu vyhľadávania v nejakom rozumnom stave. Ako sme spomínali pri opisovaní Dijkstrovho algoritmu (2.2.1), mnoho implementácií končí po nájdený hodnoty najlacnejšej cesty pre zadaný konečný vrchol. My túto ideu v tejto chvíli aplikujeme, avšak jemne pozmenenú. Budeme si počas behu Dijkstrovho algoritmu počítať, koľko krát sme už prišli do finálneho vrchola a po určitom počte jeho návštev algoritmus ukončíme. Môžeme tak spraviť vďaka zmenenej štruktúre dát, keďže konečných vrcholov náš graf obsahuje viac, každý s inou hodnotou času. Otázkou zostáva, aký bude spomínaný počet návštev konečného vrchola. Ak bude priveľký, bude algoritmus bežať zbytočne dlho. Ak zasa malý, používateľ bude nespokojný pre nízky počet výsledkov. Navyše, v prípade opätovného vyhľadávania na tých istých vstupoch len s vyššou hodnotou tohto čísla bude musieť algoritmus začínať vždy odznova. Bolo by teda rozumné, nechať používateľov samých navoliť toto číslo. Nateraz sme ale túto hodnotu napevno zadrôtovali na číslo 3. Naša implementácia algoritmu je k nahliadnutiu na obrázku 20. Treba podotknúť, že niektoré použité premenné sú definované mimo procedúry a ich význam je preto možne zistiť buď z názvu, z kontextu, alebo z prezretia si väčšej časti kódu.

```
private void DijkstrasAlgorhitm(Vertex start, string fin)
{
    updateVertecesValues();

    Dictionary<Vertex, bool> visited = new Dictionary<Vertex, bool>();
    start.parent = null;
    start.value = 0;
    DijkstrasComparator dc = new DijkstrasComparator();
    MinHeap<Vertex> inScope = new MinHeap<Vertex>(dc);
    inScope.Add(start);

    while (inScope.Count() > 0)
    {
        Vertex now = inScope.PopMin();

        List<Vertex> neighbors = graph.getNeighbors(now);
        List<Edge> incidentEdges = graph.getIncidentEdges(now);

        if (!now.name.Equals(fin))
        {
            for (int i = 0; i < neighbors.Count; i++)
            {
                Vertex v = neighbors[i];
                Edge e = incidentEdges[i];

                if (v.isThis(start)) break;
                if (visited.ContainsKey(v)) continue;

                v.addAlternatePath(e);

                int newValue = now.value + e.travellTime;

                bool incTransfers = (((now.toParent == null) || (now.toParent.name.CompareTo(e.name) != 0))
                    && (!e.waitingEdge));

                if (newValue < v.value)
                    updateVertex(v, e, newValue, now.transfers, incTransfers, inScope);
                else if (newValue == v.value)
                {
                    int newTransfers = now.transfers;
                    if (incTransfers) newTransfers++;

                    if (newTransfers < v.transfers)
                        updateVertex(v, e, newValue, newTransfers, false, inScope);
                    else if ((newTransfers == v.transfers) && v.parent.time.CompareTo(now.time) == 1)
                        updateVertex(v, e, newValue, newTransfers, false, inScope);
                }

            }
        }
        else
        {
            amountNow--;
            if (amountNow == 0) return;
        }
        visited.Add(now, true);
    }
}
```

Obr. 20: Naša implementácia Dijkstrovho algoritmu

Výsledná odpoveď algoritmu bude uložená vo vrcholoch grafu. Ich hodnoty pre požadovanú zastávku nájdeme jednoducho podľa názvu, teda vezmeme všetky vrcholy s názvom, aký je požadovaný a zoradíme ich podľa času počnúc zadaným časom pre vyhľadávanie. Pre nejaký rozumný počet týchto vrcholov necháme zrekonštruovať a do konzoly vypísať nájdené cesty. Táto hodnota by teda nemala a ani nemôže presiahnuť hodnotu návštev finálneho vrchola z predošlého odseku, momentálne teda 3, nakoľko pre ostatné vrcholy už výslednú hodnotu nepoznáme.

Vypisovanie cesty pre daný vrchol realizujeme pomocou rekurzie, čo nám zabezpečí správne poradie zastávok. Túto procedúru ukazuje obrázok 21. O samotný spôsob výpisu sa teda, ako je i zobrazené v kóde, stará iná trieda, trieda "PathShowing".

```
private void printPath(Vertex v)
{
    if (v == null) return;
    printPath(v.parent);
    if (v.parent != null)
    {
        //two verteces has been found
        //now we need to print the edge in between, if it isn't waiting
        if (!v.toParent.waitingEdge) pathShowing.printThis(v.toParent);
    }
}
```

Obr. 21: Kód starajúci sa o výpis cesty

4.5 Neefektívnosť vytvárania objektov

Program, po implementácií všetkého doposiaľ opísaného, sa zdá byť funkčný. Na testovacích vstupoch je bezchybný. To však neznamená, že funkčný naozaj je. Aplikáciu musíme, čo je aj cieľom práce, otestovať na veľkých, respektíve reálnych dátach. Po spustení na týchto dátach už ale program nepracuje tak, ako by sme očakávali. Aplikácia sa i po viac ako hodinovom čakaní nerozbieha. Využili sme preto možnosť krokovania kódu, čím sme zistili príčinu nášho problému - aplikácií trvá pridlho pretváranie dát z obrovského textového súboru na žiadané objekty. Po prezretí si kódu sme, našťastie, našli neefektívne časti, ktoré sme prerobili. Presnejšie, pri vytváraní vrcholov sme zakaždým kontrolovali či už vrchol s rovnakými údajmi náhodou neexistuje, a tak aby sme nevytvorili vrcholy nesúce rovnaké informácie. Na to sme, pri našej jednoduchej implementácií grafu, museli prejsť poľom vrcholov. Preto sme do objektu grafu popridávali rôzne slovníky, resp. hashovacie mapy, čím sa nám podarilo výrazne urýchliť vytváranie objektov. Slovníky sme dokonca využili i počas algoritmu vyhľadávania, takže aj on sa stal trocha rýchlejším. Modifikovaná implementácia grafu sa nachádza na obrázku 22.

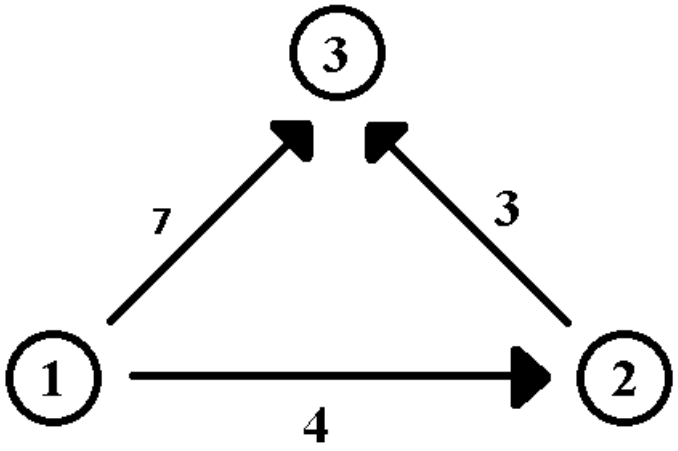
```
public class Graph
{
    internal List<Vertex> verteces;
    internal List<Edge> edges;

    internal SortedDictionary<string, List<Vertex>> allStops;
    internal Dictionary<Vertex, List<Edge>> allEdges;
    internal Dictionary<Vertex, List<Vertex>> neighbors;


    internal Graph()
    {
        verteces = new List<Vertex>();
        edges = new List<Edge>();
        allStops = new SortedDictionary<string, List<Vertex>>();
        allEdges = new Dictionary<Vertex, List<Edge>>();
        neighbors = new Dictionary<Vertex, List<Vertex>>();
    }
}
```

Obr. 22: Vylepšená implementácia grafu

Vylepšenie objektu grafu zabezpečilo, že sa aplikácia načítava približne minútu. Boli sme ju teda schopný otestovať na reálnych dátach a výsledky boli priaznivé. Dĺžka načítavania je ale stále neprajná, hoci aplikácia by mohla byť stále využiteľná, napríklad keby sa spustila a nechala bežať na serveri a používatelia by sa skrz určitú webovú stránku spytovali aplikácie svoje otázky.

4.6 Vyhľadávanie alternatívnych ciest

Posledným cieľom našej práce je implementácia nejakého vylepšenia, ktoré obohatí náš program a urobí ho príťažlivejším. Rozhodli sme sa, že medzi funkcionalitu aplikácie pridáme možnosť zobrazenia alternatívnych ciest, keďže ju postráda skoro každý iný vyhľadávací nástroj. Presnejšie, pri zobrazení výsledkov algoritmu bude pri každej zastávke možné zvoliť zobrazenie iných trás do tej zastávky s príchodom v tom istom čase.

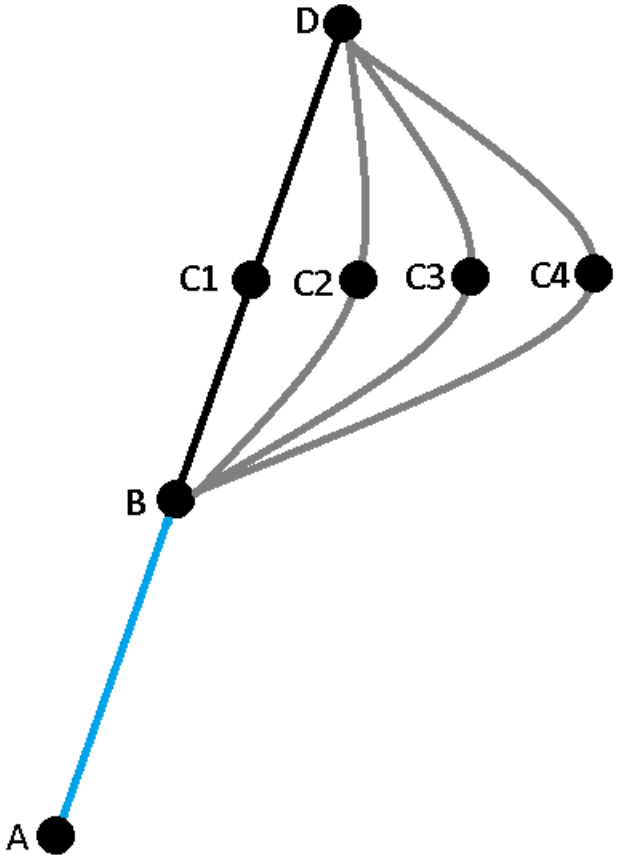


Obr. 23: Príklad alternatívnej trasy

Napríklad, pre graf zobrazený na 23 bude výsledok vyhľadávania priama cesta z vrcholu 1 do 3. Ale pri vrchole 3 v zobrazení výsledku bude napísané, že doň vedú až dve cesty, nie len tá vypočítaná. Aplikácia bude schopná na žiadosť používateľa tieto alternatívne cesty vypísať a dokonca sa i navrátiť do predošlého stavu.

Po dlhšom uvažovaní sme si uvedomili, že implementácia takéhoto vylepšenia nie je priveľmi náročná. Je ale nutné upraviť vyhľadávací algoritmus, čo ho, dúfame, príliš nezprehľadní. V prvom rade pridáme do triedy pre vrchol jedno pole hrán uchovávajúce si tie hrany, ktoré boli v Dijkstrovom algoritme skúšané ako kandidáti na najkratšiu cestu do tohto vrchola. Samozrejme, budú tam všetky - aj tie, ktorým sa nepodarilo, i tie, ktorým áno, lebo tie so zdarným porovnaním sa môžu neskôr prepísať inou, lepšou možnosťou. A ako sme už prezradili, úprava vo vyhľadávacom algoritme tiež nebude markantná. V ňom nám stačí pred porovnaním pridať hranu do spomínaného poľa pre cieľový vrchol daného vyhodnocovania.

Idea je nasledovná: Uvažujme alternatívne cesty do vrcholu v. Najskôr si musíme uvedomiť, ktoré to sú. Nech do vrcholu v smeruje n hrán. Inak povedané, existuje n hrán tvaru xv pre ľubovoľný vrchol x. Potom vieme nájsť n alternatívnych ciest do zvoleného vrcholu, pre každú takúto hranu jednu, pričom každá cesta obsahuje ako poslednú hranu hranu xv pre nejaký vrchol x spojenú s najlacnejšou cestou do vrcholu x z počiatočného vrcholu. Jej existenciu zaručuje Dijkstrov algoritmus, ktorý počíta všetky najlacejšie cesty z počiatočného vrcholu do všetkých vrcholov.



Obr. 24: Príklad alternatívnych trás

Po aplikovaní vylepšenia nastanú v Dijkstrovom algoritme len nepatrné zmeny. Pridanie prvku do poľa je z hľadiska časovej zložitosti operácia s amortizovane konštantnou časovou zložitosťou, a teda nemení celkovú časovú zložitosť algoritmu. Výpis nájdených alternatívnych ciest je však trochu zložitejší. Uvažujme, že budeme vypisovať alternatívne trasy do vrchola D na obrázku 24. Je zrejmé, že hrán, ktoré smerujú do vrchola D môže byť v najhoršom prípade n (kde n je počet hrán grafu) a taktiež dĺžka cesty medzi vrcholmi A a B môže mať dĺžku približne n. Samozrejme, tieto dve skutočnosti nemôžu nastať súčasne, avšak ak budú obi dve hodnoty blízke $n/2$, výpis bude aj tak prebiehať v časovej zložitosti $O(n^2)$. Toto je však, našťastie, veľmi okrajový prípad, keďže uvažuje nájdenú cestu naprieč všetkými hranami grafu a navyše i v nej špecifickú postupnosť hrán.

V našom modeli, kde je zastávka reprezentovaná názvom a časom, bude v priemernom prípade smerovať do nejakého vrcholu iba zopár hrán. Problémom o redukovaní nájdených alternatívnych ciest v prípade ich priveľkého počtu sa preto nebudeme zapodievať. Jednoducho ich vypíšeme všetky. Avšak výsledky pred výpisom utriedime podľa

dĺžky cesty do predposledného vrcholu, čím umiestnime najrelevantnešie výsledky navrch. Toto sčasti rieši i prvý spomínaný problém.

Implementácia vylepšenia sa naozaj nezdá byť moc pracná, jej účel je ale naozaj prínosný. Avšak nie je pravda, že jej naprogramovanie je také jednoduché. S touto implementáciu sa nám podarilo získať výsledky, ktoré, žiaľ, nevieme vypísať. Vo výpise sa ale nachádza tvrdý oriešok. Keďže potrebujeme vypísať alternatívne cesty, potrebujeme nielen zmazať predošlý výpis výsledkov, ale si aj tento stav zapamätať, keďže sa k nemu bude chcieť používateľ pravdepodobne vrátiť. Ďalej, ak si zvolíme zobraziť alternatívne cesty pre nejakú zastávku, možno si budeme chcieť opäť z vypísaného zvoliť ďalšiu alternatívnu cestu. Výpis by teda mal fungovať do ľubovoľnej hĺbky. Ako nádherné riešenie tohto problému sa nám ponúka použiť návrhový vzor *Memento*, ktorého *Caretaker* si bude postupne zapamätávať stavy a na žiadosť nám posledný zapamätaný stav poskytne a zároveň ho odstráni zo svojej pamäte. Na jeho pamäť zasa poslúži dátová štruktúra *zásobník* a ukladané stavy budú reprezentované triedou so zoznamom zoznamov hrán (zoznam hrán reprezentuje cestu a zoznam týchto ciest zas zobrazovaný výsledok). Implementácia *Mementa* nám teda s prehľadom vyriešila všetky starosti.

Záver

V práci sme skúmali vlastnosti a popisovali algoritmy vhodné na vyhľadávanie spojení hromadnej dopravy. Zamerali sme sa na tri základné algoritmy, ktoré sme medzi sebou v mnohých ohľadoch porovnávali, vďaka čomu sme dokázali usúdiť, ktorý z nich bude najvhodnejším kandidátom na spomenuté vyhľadávanie. Porovnávali sme ich časovú zložitosť, ich použiteľnosť na malých i veľkých vstupoch, resp. na vstupoch s malým či veľkým počtom hrán. Najlepšie obstál Dijkstrov algoritmus, ktorý sa zdá byť najvšeobecnejší a najefektívnejší pri použití ľubovoľnej grafovej štruktúry. Samozrejme, ako sa spomína aj v [3], existujú rôzne zložitejšie algoritmy, ktoré sú oveľa rýchlejšie na vyhľadávanie spojení MHD. Mnoho výskumov a návrhov komplikovanejších algoritmov zhrnuli a porovnali Fan a Machemehl v svojom diele [9].

So znalosťou Dijkstrovho algoritmu sme dokázali vytvoriť softvér, ktorý na poskytnutých dátach nájde najlacnejšie cesty medzi zadanými vrcholmi. Program si najprv z dát vytvorí grafovú štruktúru, s ktorou následne pracuje. Program je navrhnutý tak, že vďaka vstupu od požívateľa vo forme názvu počiatočnej a konečnej zastávky dokáže uskutočniť vyhľadávanie a následne vypísať vypočítané výsledky v prívetivej forme. Aplikáciu sme otestovali i na dátach Bratislavskej hromadnej dopravy, pričom nás prekvapila rýchlosť algoritmu, ktorému i pri tak obrovskom počte zastávok a liniek trvalo nájsť riešenie zakaždým menej ako sekundu. Samozrejme tomu prispieva prítomnosť ukončenia vyhľadávania, keď už riešenie obsahuje dostatočne veľa potrebných informácií.

Navrhli sme a implementovali zobrazovanie alternatívnych trás do danej zastávky. Takéto vylepšenie zväčša iné implementácie podobného charakteru neposkytujú, a preto sme sa preň rozhodli.

Na um nám prichádzajú i možné pokračovania práce. Napríklad by sa dalo spracovať i iné, pokročilejšie algoritmy a na základe ich spoločného porovnania vybrať lepší algoritmus ako Dijkstrov na vyhľadávanie spojení hromadnej dopravy. Ďalšou možnosťou je implementovať viacero algoritmov na vyhľadávanie a tie na základe rozumných meraní medzi sebou porovnávať. Taktiež by sa dalo implementovať iné použiteľné vylepšenia a zakomponovať ich do kódu našej aplikácie, prípadne zlepšiť či zefektívniť terajšie riešenie, napríklad sprehľadnením či vizuálnym vylepšením používateľského rozhrania.

Literatúra

[1] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[2] Michael J Bannister and David Eppstein. Randomized speedup of the bellman–ford algorithm. In *2012 Proceedings of the Ninth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 41–47. SIAM, 2012.

[3] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *European Symposium on Algorithms*, pages 290–301. Springer, 2010.

[4] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[5] Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based public transit routing. *Transportation Science*, 49(3):591–604, 2014.

[6] Reinhard Diestel. Teória grafov, 2000. [Citované 2017-12-4] Dostupné z `http://www.dcs.fmph.uniba.sk/~haviarova/uktg/#materialy`.

[7] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96:59–69, 2014.

[8] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

[9] Wei Fan and Randy B Machemehl. Optimal transit route network design problem: Algorithms, implementations, and numerical results 6. performing organization code. 2004.

[10] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental and Efficient Algorithms*, pages 319–333. Springer, 2008.

[11] Takahiro Ikeda, Min-Yao Hsu, Hiroshi Imai, Shigeki Nishimura, Hiroshi Shimoura, Takeo Hashimoto, Kenji Tenmoku, and Kunihiko Mitoh. A fast algorithm for finding better routes by ai search techniques. In *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pages 291–296. IEEE, 1994.

[12] Jana Katreniaková. Vizualizácia a kreslenie pekných grafov, 2014. [Citované 2017-12-4] Dostupné z `http://sccg.sk/~ferko/VizualizaciaPreGrafikov.pdf`.

[13] S Meena Kumari and N Geethanjali. A survey on shortest path routing algorithms for public transport travel. *Global Journal of Computer Science and Technology*, 9(5):73–75, 2010.

[14] Patrick Lester. A* pathfinding for beginners. *online]. GameDev WebSite. http://www. gamedev. net/reference/articles/article2003. asp (Acesso em 08/02/2009)*, 2005.

[15] Timothy Merrifield. *Heuristic Route Search in Public Transportation Networks.* PhD thesis, University of Illinois at Chicago, 2010.

[16] Jakub Novák. Dynamická navigácia osôb so spätnou väzbou v mestskej hromadnej doprave. Bakalárska práca, Univerzita Komenského v Bratislave, 2016.

[17] José Luis Santos. k-shortest path algorithms. 2007. [Citované 2017-12-4] Dostupné z `https://estudogeral.sib.uc.pt/jspui/bitstream/10316/11305/1/k-Shortest%20path%20algorithms.pdf`.

[18] Dominik Schultes and Peter Sanders. Dynamic highway-node routing. In *International Workshop on Experimental and Efficient Algorithms*, pages 66–79. Springer, 2007.

[19] Pavol Ďuriš. *Tvorba efektívnych algoritmov.* Knižničné a edičné centrum FMFI UK, 2009.

[20] Kari Edison Watkins, Brian Ferris, and G Scott Rutherford. Explore: An attraction search tool for transit trip planning. *Journal of Public Transportation*, 13(4):6, 2010.