Korszerű számítástechnikai módszerek a fizikában 1

Projekt jegyzőkönyv

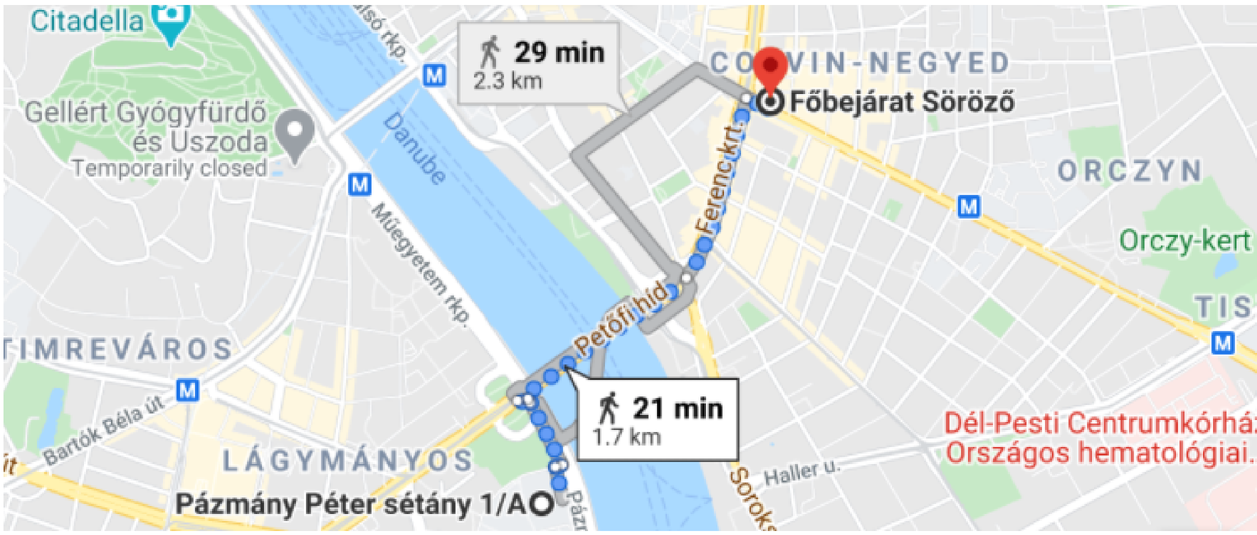
Szűcs Máté

Legrövidebb út algoritmus

Eötvös Loránd Tudományegyetem 2021

1. Bevezetés

Az útvonaltervezéssel mindennap találkozunk, például iskolába vagy munkába menet szeretnénk mindig a legoptimálisabb, azaz legrövidebb úton eljutni. Mivel az időnket mindennél többre becsüljünk, ezért mindig törekszünk a legrövidebb idő alatt eljutni a célunkba.



1. ábra. Egy példa az útvonaltervezésre Google Maps-ben

A legrövidebb utak megtalálására általában különféle alkalmazásokat hívunk segítségül, mint a Google Maps, Waze vagy Apple Maps. A Google Maps útvonal keresőjére láthatunk példát az 1 ábrán. Ezek a szolgáltatások különbőző, az informatikában található útvonalkereső algoritmusokat használnak, hogy megtalálják számunkra a legkedvezőbb útvonalakat. A Google Maps alapja a Dijkstra algoritmus.

Dijkstra legrövidebb út algoritmusával megtalálhatjuk a legrövidebb utakat adott csúcsok között egy súlyozott gráfban, amik reprezentálhatnák például egy úthálózatot [1]. A holland informatikus Edsger W. Dijkstra [2] találta ki 1956-ban és publikálta 1959-ben [3]. Hivatása szerint elméleti fizikus volt, de programozóként dolgozott a Mathematisch Centrum-ban Hollandiában 1952-től 1962-ig.

Az évek során több változata is elterjedt az algoritmusnak, az eredeti célja, hogy megtalálja a legrövidebb utat két csúcs között. Napjainkban elterjedtebb az a változat, ahol egy kijelölt "source" csúcs és a gráfban található összes többi között keressük [4], ezt implementáltam a munkám során C++ nyelven, de könnyen belátható, hogy kis módosítással ez visszavezethető az előzőre.

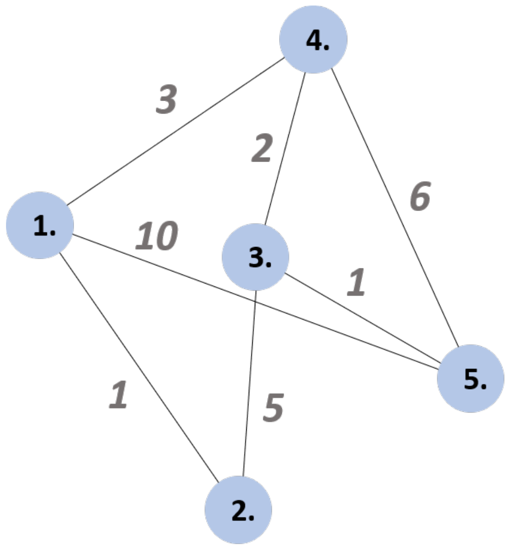
Az algoritmus komplexitása:

$$\Theta((|V| + |E|)\log|V|) \tag{1}$$

ahol $|V|$ a csúcsok száma, $|E|$ pedig az éleké.

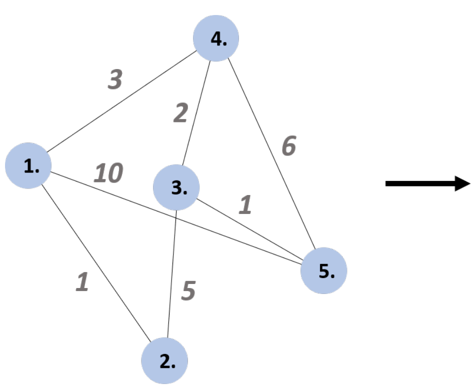
2. Az algoritmus

Hogy megértsük az algoritmust először is tekintsük a 2 ábrán látható kis súlyozott gráfot.



2. ábra. Példa gráf

A gráfnak 5 darab csúcsa van, néhány között láthatunk különféle súlyokkal ellátott éleket. Legyen a kiindulási csúcsunk az egyes számú és tekintsük azt az esetet, amikor mindegyik csúcshoz keressük a legrövidebb utat. Az első lépés, hogy ezt a gráfot valami olyan formába öntjük át, amivel könnyen tudunk dolgozni, erre a legelterjedtebb megoldás az úgynevezett szomszédsági mátrix. Szomszédsági mátrixnak nevezzük egy n csúcsú G gráfhoz tartozó $n \times n$-es mátrixot, amelynek, irányítatlan gráf esetén, a nem főátlóban szereplő a_{ij} eleme az i és j csúcsokat összekötő él súlya. Ha két csúcs között nincs él, akkor a súly értékét nullának vesszük, így van ez a főátló a_{ii} elemeinél is, hiszen most nem engedünk meg hurkokat a gráfban.



$$\begin{pmatrix} 0 & 1 & 0 & 3 & 10 \\ 1 & 0 & 5 & 0 & 0 \\ 3 & 0 & 0 & 2 & 1 \\ 3 & 0 & 2 & 0 & 6 \\ 10 & 0 & 1 & 6 & 0 \end{pmatrix}$$

3. ábra. Gráfból szomszédsági mátrix

A továbbiakban ezzel a szomszédsági mátrixszal dolgozunk. Készítünk egy n elemű *distances* listát, amiben tároljunk a legrövidebb utak értékét a *source* csúcsunktól. Válasszuk *source*-nak az elsőt, tehát a listában az adott indexekhez tartozó értékek, a legrövidebb utak súlyainak összege az első, vagyis nulladik indexú csúcstól az adott indexű csúcsig. Kezdetben mindegyik értéket végtelenre inicializáljuk a listában, a *source*-hoz tartozót pedig nullának. Ezután készítünk egy *queue* párokból álló listát, ahol a párok első eleme a súly, a második pedig a csúcs indexe. Kezdetben csak egy párt tartalmaz, a *source* csúcsot, ahol nullára állítjuk a súlyt és a csúcs indexe lesz a második elem, esetünkben a nulla. Végül inicializálunk egy *parents* listát, hogy később számon tudjuk tartani az útvonalat magát.

Ezután következik a legrövidebb út keresése. A műveleteket egy folytonos ciklusban végezzük el, ami addig fut, amíg a *queue* üres nem lesz. Először is megkeressük az adott csúcsunktól a *queue*-ban lévő legközelebbi elemet, azt vesszük a jelenlegi csúcsnak, tehát úgymond "átugrunk" rá legyen ez az *u* csúcs, és az előzőt töröljük a *queue*-ból. Végigmegyünk a szomszédsági mátrixnak az *u* csúcsunkhoz tartozó során és mindegyikre megnézzük, hogy rövidebb-e ez az út a *source* csúcsból oda, ha az *u* csúcson keresztül megyünk. Ha azt találjuk a szomszédsági mátrix adott csúcsára, hogy a hozzátartozó, eddig a *distances* listában szereplő érték nagyobb, mint az *u* csúcshoz tartozó érték és a két csúcs közötti él súlyának összege, akkor a következőket tesszük:

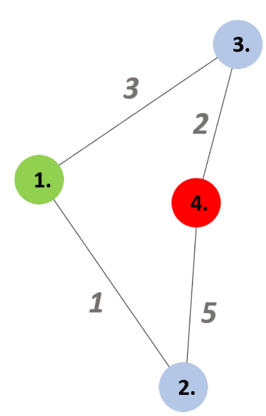
1. Megnézzük, hogy ebben a csúcsban jártunk-e már, tehát, hogy az értéke végtelen-e, ha igen akkor töröljük a *queue*-ból. Hiszen lehet, hogy korábban már eljutottunk ide egy másik csúcsból és hozzáadtuk a *queue*-hoz (lásd 3. lépés), de most annál rövidebb utat találtunk, ezért az előzőt eltávolítjuk.

2. Frissítjük a hozzátartozó értéket a *distances*-ben, *u* + *súly*-ra;

3. Végül hozzáadjuk a *queue*-hoz ezt az új talált csúcsot.

Itt láthatjunk, hogy a legelőször említett folytonos ciklusunk sose lesz végtelen, hiszen csak akkor adunk hozzá elemet a *queue*-hoz, ha rövidebb utat találtunk, amikből pedig nem áll rendelkezésre végtelen számú.

Egy fontos dolog, amit észrevehetünk az algoritmus leírásából, hogy mindegyik csúcsnál a legközelebbit választja a következőnek, tehát az adott pillanatban a legoptimálisabb lépést hajtja végre. Az ilyen algoritmusokat *greedy*-nek nevezzük, ezek minden lépésnél a lokális legoptimálisabb döntést hozzák és nem veszik figyelembe az egész problémát. Így az algoritmus sokkal gyorsabban lefut mintha az összes élt vizsgálnánk, de lehet, hogy összességében nem a legoptimálisabb útvonalat találjuk meg. Tekintsük a a 4 ábrán látható gráfot.



4. ábra. Greedy algoritmus

Tegyük fel, hogy az első csúcsból szeretnénk eljutni a negyedikbe, ezt a másodikon vagy a harmadikon keresztül tehetjük meg. Egy *greedy* algoritmus az első lépésnél a második csúcsot fogja választani, hiszen ahhoz kisebb súly tartozik, mint a hármashoz ($1 < 3$), viszont láthatjuk, hogy összességében a hármason keresztül lenne rövidebb az út hiszen: $3 + 2 < 1 + 5$.

Miután kiléptünk a folytonos ciklusból, a *distances* listánk tartalmazza a legrövidebb utak értékét és a *parents* listából visszagöndölíthetjük magát az útvonalat, hiszen az adott indexéhez tartozó érték annak a csúcsnak a sorszáma ahonnan az indexhez tartozó csúcshoz léptünk.

A bevezetésben említett változatát az algoritmusnak, amikor is két adott csúcs között keressük a legrövidebb utat, is megkaphatjuk ezekből a lépésekből egy kicsi módosítással. Az egyetlen változtatás amit tennünk kell, hogy az algortimus elején, miután kiválasztottuk az *u* csúcsunkat, vizsgáljuk meg, hogy ennek a sorszáma megegyezik-e az elérni kívánt *target*-éval. Ha igen, akkor egy `break` utasítással kiléphetünk a folytonos ciklusból és vége is a keresésnek.

3. Gondolatok az implementálásról

Mivel a kurzus fontos szerepet szánt a modern objektumorientált programozásnak és, mert úgy érzem, hogy ez a gyengeségeim közé tartozik, ezért úgy döntöttem, hogy objektumorientáltan írom meg az algoritmust. Ebben a fejezetben a technikai oldalról szeretnék bemutatni néhány dolgot, amik a *C++*-beli implementálás során kerültek elő.

Ahogy az előadások során tanultuk, a kódomat kétfelé osztottam, egy `graph.hpp` tartalmazza használt függvényeket és osztályokat, a `main.cpp` és `toy.cpp` pedig az algoritmus lefuttatását két különböző gráfra. Az elsőben található a nagyobbik, melyet a *Code #LikeABosch* nevű challenge első fordulójából kölcsönöztem [5]. Az utóbbi pedig egy kisebb, amit az előző fejezetben láthattunk.

A legrövidebb út algoritmushoz először is szükségünk van egy gráfra, ehhez írtam meg a `Graph` osztályt. `private` attribútumai a csúcsok száma és a szomszédsági mátrixa, amely egy `list<pair <int, int>>` tipusú tároló, a párok első elemei a csúcsok a második pedig a súlyok. Láthatjuk, hogy itt most a szomszédsági mátrix sorai nem szigorúan listák, hanem párokból áll. A `public` metódusok pedig az a `Graph`, amivel meghívhatjuk magát az osztályt itt meg kell adnunk a csúcsok számát. A `add_edge` függvény, amivel a csúcsok közé éleket tehetünk. A `print_adj` fügvénnyel kiírhatjuk a gráf szomszédsági mátrixát, végül a `Dijkstra` metódus elvégzi a Dijkstra algoritmust a gráfon egy *source* és egy *target* csúcsot megkapva. Ez a metódus meghívja a `Display_dijkstra` függvényt, ami kiírja nekünk a legrövedibb talált útvonalat a két csúcs között és a súlyok összegét.

Az implementálás során a *C++ standard library* több elemét is használtam például: `list`, `vector` , `pair` stb. De a leghasznosabb a `set` volt. Ahogy láthattuk az előző fejezetben, az algoritmus egyik első lépésénél meg kell keresnünk az adott csúcstól a *queue*-ban lévő legközelebbi csúcsot. Ehhez írhatnánk valamilyen függvényt is, amit felhasználva megtalálhatnánk, de a `set` segítségével egyszerűbb. Ez egy olyan tároló, amelyben az elemek valamilyen kulcs szerint vannak sorban [6], esetünkben a kulcs a távolság volt, így a *queue* első eleme mindig a legkisebb távolsággal rendelkező csúcs lehetett, elég volt az eltávolítanunk.

4. Diszkusszió

Összességében azt mondhatom, hogy élveztem a munkát a nagy projektemen, örülök, hogy a tárgy oktatói ezt a rendszert választották. Úgy gondolom, hogy a félév során sokat tanultam a *C++* programozásról és a projektnek hála már nem olyan idegen tőlem az objektumorientált programozás se.

Hivatkozások

[1] „Wikipedia: Dijkstra's algorithm."
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.

[2] „Wikipedia: Edsger W. Dijkstra."
https://en.wikipedia.org/wiki/Edsger_W._Dijkstra.

[3] E. Dijkstra, „A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[4] K. Mehlhorn and P. Sanders, „Shortest paths," in *Algorithms and Data Structures*, ch. 10, pp. 191–215, Berlin: Springer, Berlin, Heidelberg, 2008.

[5] „Code likeabosch automotive challenge."
https://codelikeabosch.mphacks.hu/.

[6] „Cppreference: std::set."
https://en.cppreference.com/w/cpp/container/set.